POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Automated Assembly-Level Mitigation of Transient Execution Attacks using Diversified Code Variants and Genetic Search

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Gabriele Giordanelli**

Student ID: 10934859
Advisor: Prof. Luca Maria Cassano
Co-advisors: Elia Lazzeri
Academic Year: 2025-26

Abstract

Transient Execution Attacks (TEAs) such as Spectre and Meltdown exploit speculative and out-of-order execution to transiently access sensitive values and leak them through microarchitectural side channels. Although these transient instructions are eventually squashed and leave no architectural traces, they can still influence microarchitectural state, enabling attackers to recover secrets through timing measurements. Practical defenses must thus reduce exploitable transient behavior while keeping performance overhead and deployment complexity acceptable, especially on already-deployed processors.

This thesis presents a software-only, compile-time mitigation framework based on assembly-to-assembly rewriting of x86-64 compiler-emitted code. The framework detects candidate transient "windows" for three TEA classes—Spectre V1 (bounds-check bypass), Spectre V4 (speculative store bypass), and Meltdown-like patterns—and rewrites only the affected regions. For each window, it generates multiple semantically equivalent hardened variants by composing local transformations (speculation-stopping fences, data sanitization/masking, dependency injection, and structural perturbations), and injects a lightweight runtime selector to diversify execution and reduce gadget repeatability. Because this configuration space is large, the framework includes a genetic optimizer that, using a dataset-driven surrogate evaluation loop, searches for a single global configuration that maximizes protection under an overhead budget, or minimizes overhead under a protection target. An experimental campaign on three TEA proofs of concept (PoCs) shows that diversification alone tends to reduce leakage as variant multiplicity increases; for instance, in Spectre V1 the duplication-only baseline reduces leakage from 97.8% at $N = 1$ to 86.8% at $N = 75$, with a 32.69% cycle overhead. However, stronger suppression typically requires combining at least one direct mitigation mechanism with lighter perturbations. Accordingly, maximizing protection under a 24% overhead budget yields a configuration that guarantees at least 95.74% worst-case protection across all three targeted attacks.

Keywords: transient execution attacks; spectre; meltdown; assembly rewriting; software-only mitigation; code diversification; genetic optimization.

Abstract in lingua italiana

Gli attacchi a esecuzione transiente (Transient Execution Attacks, TEAs) come Spectre e Meltdown sfruttano l'esecuzione speculativa e out-of-order per accedere in modo transiente a valori sensibili e farli trapelare tramite canali laterali microarchitetturali. Sebbene queste istruzioni transienti vengano annullate e non lascino tracce architetturali, possono comunque influenzare lo stato microarchitetturale, consentendo agli attaccanti di recuperare i segreti tramite misure di timing. Le difese pratiche devono quindi ridurre il comportamento transiente sfruttabile mantenendo al contempo accettabili l'overhead prestazionale e la complessità di distribuzione, soprattutto su processori già in uso. Questa tesi presenta un framework di mitigazione esclusivamente software, applicato a tempo di compilazione, basato sulla riscrittura assembly-to-assembly di codice x86-64 emesso dal compilatore. Lo strumento rileva "finestre" transienti candidate per tre classi di TEA—Spectre V1 (bounds-check bypass), Spectre V4 (speculative store bypass) e pattern di tipo Meltdown—e riscrive solo le regioni interessate. Per ciascuna finestra genera molteplici varianti semanticamente equivalenti componendo trasformazioni locali (fence che arrestano la speculazione, sanitizzazione/mascheramento dei dati, iniezione di dipendenze e perturbazioni strutturali) e inserisce un selettore runtime per diversificare l'esecuzione e ridurre la ripetibilità dei gadget. Poiché questo spazio di configurazione è ampio, il framework include un ottimizzatore genetico che, tramite un ciclo di valutazione surrogato basato su un dataset sperimentale, ricerca una singola configurazione globale che massimizzi la protezione sotto un vincolo di overhead oppure che minimizzi l'overhead sotto un obiettivo di protezione. Una campagna sperimentale su tre proof of concept (PoC) di TEA mostra che la sola diversificazione tende a ridurre la fuga d'informazione all'aumentare della molteplicità delle varianti; ad esempio, in Spectre V1 la baseline basata esclusivamente sulla duplicazione riduce la fuga d'informazione dal 97.8% a $N = 1$ all'86.8% a $N = 75$, con un overhead in cicli pari al 32.69%. Tuttavia, una soppressione più efficace richiede in genere di combinare almeno un meccanismo di mitigazione diretto con perturbazioni più leggere. Di conseguenza, massimizzare la protezione sotto un vincolo di overhead del 24% produce una configurazione che garantisce almeno il 95.74% di protezione nel caso peggiore su tutti e tre gli attacchi considerati.

Parole chiave: attacchi a esecuzione transiente; spectre; meltdown; riscrittura assembly; mitigazione solo software; diversificazione del codice; ottimizzazione genetica.

Contents

1 | Introduction

Over the past decades CPU performance has been continuously improved by shrinking processing technology and increasing clock speeds. When physical limitations started hindering this approach, industry and academia shifted their focus towards increasing the number of cores and optimizing instruction pipelines. To achieve this, hardware architects have implemented aggressive microarchitectural optimizations that allow the CPU to improve its performance by executing operations for subsequent instructions ahead of time or even out-of-order [28][40], before their architectural validity is confirmed. The fundamental assumption behind this design is the isolation provided by the Instruction Set Architecture (ISA), which, in practice, acts as an abstract contract guaranteeing that only correctly resolved instructions can update the architectural state (registers and main memory). If a speculation goes wrong, the processor rolls back, discarding any architectural effects of the mis-speculated instructions. Such instructions execute *transiently*, a phenomenon known as *transient execution* [29]. Unfortunately, recent research has exposed an important flaw in this abstraction: while the architectural state is rolled back, the microarchitectural state (such as the cache hierarchy, Pattern History Table (PHT) and Branch Target Buffer (BTB)) still reflects the side effects generated during the transient execution [15][9]. This flaw has given rise to a class of vulnerabilities known as *Transient Execution Attacks* (TEAs), exemplified by Spectre and Meltdown [29][30]. In these scenarios, the attack exploits the processor, letting it execute transient instruction sequences which access sensitive data. Although these instructions are eventually discarded, the secret data leaves footprints in the microarchitectural state (e.g. cache state), which can be later recovered at the architectural level via a side channel [38]. These attacks bypass software-level isolation mechanisms, including kernel-user separation and virtual memory protections. Consequently, effective software-based defenses are needed to mitigate TEAs on already deployed processors, providing continuously updatable protection that can evolve as new transient-execution variants are discovered in the future.

1.1. Problem statement

The central problem addressed in this thesis is the unavoidable trade-off between preserving the performance benefits of modern microarchitectures and providing effective security against transient execution vulnerabilities [9]. Unlike conventional software flaws, TEAs can arise even in correct code: an attacker can abuse speculative control-flow and memory-dependence mechanisms to steer execution into short *transient windows*. Mitigation is therefore not about fixing a single defect, but about systematically constraining or decoupling classes of transient behaviors that are side effects of intended hardware optimizations[15].

From a software-engineering perspective, this is challenging. Strong countermeasures such as speculation barriers or strict ordering constraints can close transient windows, but they often lead to high overhead if applied broadly or conservatively [1]. On the contrary, narrowly scoped defenses require reliably identifying where exploitable windows occur [9], despite compiler-emitted patterns, complex control flow, and differences across TEA classes. Practical defenses must thus be usable on already deployed processors, scale to large binaries at compile time, and remain tunable to control the security–overhead trade-off. Instead, current software-based mitigations for TEAs usually suffer from one or more of the following issues:

- **High performance overhead under conservative hardening**: comprehensive defenses can be very expensive in terms of performance overhead when applied broadly.
- **Incomplete/variant-driven coverage**: many mitigations are tailored to specific TEA patterns and do not generalize across attack classes.
- **Limited scalability and tunability**: effective hardening is often manual and offers limited control over the security-overhead trade-off at scale.

As a result, there is a need for systematic mitigation approaches that are scalable at compile time, which provide broader coverage across TEA classes, and expose explicit mechanisms to control the security–overhead trade-off rather than forcing a fixed point.

1.2. Contributions of the Thesis

This thesis proposes and implements a novel software-only mitigation framework, based on software manipulation, that detects transient execution vulnerable code patterns, rewrites

them into hardened variants and optimizes the security-performance trade-off via genetic search. The key contributions of this work include:

- **End-to-end mitigation framework**: a software-only framework built for TEA mitigation that hardens vulnerable code regions using automated transformation and configurable rewriting strategies.
- **Variant-based hardening workflow**: a variant-based hardening workflow integrated into the framework which enables the generation of alternative mitigation implementations for the same vulnerable region.
- **Genetic optimization module**: a built-in optimization strategy for selecting mitigation variants under a security or performance overhead objective using the genetic algorithm.
- **Experimental artifacts**: an experimental evaluation that benchmarks three TEA variants and reports security-overhead trade-offs for baseline and genetically optimized mitigation configurations.

Taken together, these contributions outline the goals of the framework: use software manipulation to apply multiple hardening transformations as semantically equivalent variants, making the security-overhead trade-off explicit and aiming for a broader coverage across TEA classes without sacrificing scalability and more performance than strictly necessary.

Due to the complexity of the pipeline, it is helpful to have a very high-level view of each stage, in order to have mental clarity as soon as possible on how the framework is structured (this overview will purposely exclude the genetic optimization and will focus only on the main stages concerning software manipulation). The pipeline is structured as follows:

1. **Input parsing**: the framework receives as input an assembly file, it then parses and saves each instruction and/or directive in appropriate data structures for later use.
2. **Detection**: the framework accesses the previously mentioned data structures storing the contents of the input, and detects possible transient windows.
3. **Variants generation**: for each detected window, the framework generates a certain number of semantically equivalent, but structurally different variants based on the user's request and on the chosen parameters.
4. **Output writing**: the framework writes a new, mitigated assembly file which is

ready to be compiled and run safely.

The remainder of the thesis expands each stage in detail, from the design choices to the implementation, as well as the evaluation methodology.

The resulting security–performance trade-off is evaluated experimentally using end-to-end leakage (% bytes stolen) and cycle overhead, comparing duplication-only baselines against hand-picked and optimized multi-transformation configurations. Across three TEA proof-of-concept attacks [13][34][16], diversification via window duplication already reduces leakage as variant multiplicity increases (e.g. for Spectre V1 the duplication-only baseline drops from 97.8% bytes stolen at N=1 to 86.8% at N=75, with a 32.69% cycle overhead). However, diversification alone is not uniformly effective: for Spectre V4 it reaches 0% leakage only at very large N (e.g., N=5000) and with a high overhead (48.34%), and for Meltdown it still leaks 83.21% at N=5000 despite only 11.28% overhead. Under an explicit overhead budget, the genetic optimizer selects a single global configuration that achieves 95.74% worst-case protection across Spectre V1, Spectre V4, and Meltdown-like patterns while remaining within a ≈24% overhead envelope; notably, on Spectre V4 it improves over the best hand-picked baseline (95.93% vs 87.57%) at lower overhead (24.02% vs 29.21%).

1.3. Thesis organization

The structure of the thesis is organized as follows:

- **Chapter 2: Background**
  This chapter outlines the theoretical foundations necessary to understand the topic of the thesis, detailing architectural mechanisms that enable TEAs and the recurring vulnerability patterns they exploit. It also concludes by defining the threat model and the security objectives that guide the remainder of the thesis.

- **Chapter 3: Related works**
  This chapter analyzes the state of the art in TEAs software-based mitigation techniques grouping them in families and discussing their core ideas, strengths, limitations and applicability.

- **Chapter 4: Proposed technique and implementation**
  This chapter presents the implementation of the proposed technique, explaining in detail the pipeline (from parsing to rewriting) and the proposed mitigation strategies developed to reduce transient leakage.

- **Chapter 5: Genetic optimization**
  This chapter motivates the use of a genetic algorithm [24] to balance security and performance overhead and details the proposed optimization strategy used to select efficient mitigation variants.
- **Chapter 6: Experimental analysis**
  This chapter presents the experimental work carried out using the tool, analyzing the results obtained with three different variants of TEAs along with the optimized results provided by the genetic algorithm.
- **Chapter 7: Conclusions and future developments**
  This chapter summarizes the strengths and weaknesses of the current implementation and proposes plans and ideas for possible future expansions of the framework.

2 | Background

This chapter provides the reader with the theoretical knowledge needed to understand TEAs, their foundational concepts, and why they work. It is organized into three sections. The first is dedicated to the microarchitectural foundations that enable speculation. The second treats microarchitectural side channels, formalizes a generic TEA model and discusses the three attack classes targeted in our evaluation. The third outlines the threat model considered throughout the thesis.

2.1. Microarchitectural foundations enabling TEAs

At their core, TEAs are enabled by the discrepancy between the processor's architecturally defined execution model and the speculative microarchitectural mechanisms used to maximize performance, for this reason, it is useful to begin by reviewing the microarchitectural foundations that make these attacks possible.

2.1.1. Instruction Set Architecture and Microarchitecture

To understand how the optimizations introduced to processors enable TEAs, it is necessary to distinguish between the two primary abstraction layers of a processor: the Instruction Set Architecture and the microarchitecture.

The *Instruction Set Architecture* (ISA) provides an interface between hardware and software, it defines the instructions that a processor supports, the available registers, the addressing mode and describes the execution model [9]. The *microarchitecture* instead is the concrete implementation of the ISA, it describes how it is implemented in the processor in terms of pipeline depth, execution units, cache hierarchies and branch prediction mechanisms.

The critical distinction to make for our analysis of TEAs is that the microarchitecture maintains a state - *the microarchitectural state* - which is a superset of the architectural state defined by the ISA [15]. While the architectural state (e.g. register contents and

main memory) is visible to the programmer, the microarchitectural state (e.g. cache lines, branch history buffers and reorder buffers) is designed to be invisible, as its job is primarily to facilitate performance optimizations like speculation.

2.1.2. Out-of-order execution essentials

The fundamental goal of Out-of-Order (OoO) execution is to maximize the Instruction-Level-Parallelism (ILP) to prevent processor execution units from staying idle. In strict in-order execution, a high-latency operation (such as a cache miss or a complex floating-point computation) stalls the pipeline, preventing subsequent independent instructions from executing.

To overcome this, modern microarchitectures decouple execution stage from fetch and commit stages. This decoupling allows the processor to look ahead and schedule instructions to idle execution units as soon as their respective operands are available, regardless of the original order in which the instructions were. While this has a positive result in terms of throughput, it needs speculative execution, instructions are processed before it is "certain" (architecturally speaking) that they should be executed or that their results are valid [1]. This speculative processing creates a transient execution window because instructions modify the microarchitectural state before they are committed to the architectural state or discarded.

Microarchitectural structures (ROB)

The implementation of OoO execution relies heavily on the Reorder Buffer (ROB) and the Unified Reservation Station (Scheduler). Instructions are fetched, decoded into micro-operations (μOPs) in program order and then dispatched to the Reservation Station. The Reservation Station queues operations and then proceeds to dispatch them to execution units once their corresponding data dependencies are resolved, performing a conversion of the sequential instruction stream into a data-flow graph. The ROB is allocated in program order at dispatch time. It is used as a holding buffer for instruction results that have completed execution but have not retired yet.

This structure acts as a bridge between the out-of-order execution backend and the in-order commitment needed by the architectural side to achieve correct program semantics [3]. To eliminate false data dependencies (Write-After-Write and Write-After-Read hazards), register renaming maps architectural registers to a bigger pool of physical registers, allowing independent instructions to proceed in parallel [40].

Recovery and rollback

The ROB is responsible for ensuring that instructions retire (commit their results to the architectural state) in strict program order. If an instruction at the front of the ROB generates an exception or is flagged as a part of a mispredicted path, the processor must start a rollback. The ROB flushes the pipeline, discarding the results of all younger, transient instructions that are currently present in the buffer.

Recovery mechanisms, similar to those described in Checkpoint Processing and Recovery (CPR), may utilize periodic checkpoints of the mapping table to restore efficiently the architectural state [3]. While this process restores the register file and Program Counter (PC) to a valid state, basically making the transient instructions architecturally invisible, the microarchitectural side effects (such as cache line allocations) persist, and they are responsible for the creation the side channels which can be exploited by TEAs.

2.1.3. Branch prediction essentials

Branch prediction is needed by deep pipelines given the high frequency of control-flow instructions present in modern software. Without prediction, the processor would stall at every conditional or indirect branch until the target address and condition are resolved, resulting in severely degraded performance. To maintain a full pipeline, the fetch unit speculatively predicts the direction and the target of branches cycles before the branch instruction gets actually executed.

This optimization creates a speculative execution path. When the predictor is accurate, the performance is maximized, however, a misprediction will result in the execution of instructions along a "wrong path". These instructions form a transient window where the processor executes code that is invalid architecturally, accessing data and modifying microarchitectural state before the misprediction is correctly detected.

Microarchitectural structures (BTB, PHT, RSB)

Modern branch prediction units (BPUs) utilize complex structures to predict both branch direction and targets. The Branch Target Buffer (BTB) acts as a cache for branch destination addresses, thanks to this the fetch unit can determine the next PC before the instruction is decoded. For conditional branches, the Pattern History Table (PHT) stores prediction state (often using saturating counters) which are indexed by global or local branch history registers.

Sophisticated predictors, such as TAGE or Perceptron-based predictors, use long history lengths and neural networks to minimize misprediction rates [25][39]. Additionally, there is also the Return Stack Buffer (RSB) which is a specialized structure used to predict the targets of RET instructions by pushing addresses on a call and popping them on a return. These structures are often shared between threads (e.g. in SMT) or processes, making them possibly susceptible to training by an attacker [31].

Recovery and rollback

When the execution unit resolves the actual outcome of a branch and detects a deviation from the prediction, a misprediction event is triggered. The processor must squash all instructions issued after the branch (the 'shadow' of the branch) and flush the pipeline [29]. The Reorder Buffer and the rename tables are correctly rolled back to the state at the branch boundary and fetching restarts from the correct path [3].

While the architectural results of the transient instructions are discarded, the microarchitectural changes like entries brought into the cache or Translation Lookaside Buffer (TLB) during the transient window, remain. Attacks like Spectre exploit this by mistraining the predictor (PHT or BTB) into intentionally creating a large transient window where secret data can be accessed and encoded into the cache before the squash happens [3].

2.1.4. Memory disambiguation & speculative loads (for V4)

Memory operations frequently dominate execution time, making the strict ordering of loads and stores a significant problem in terms of performance in superscalar processors. If a processor was forced to stall each load instruction every time a preceding store instrucition has not computed its target address yet, the pipeline would spend a lot of its time staying idle. To avoid this issue, architects employ memory disambiguation strategies that allow loads to issue speculatively, bypassing older stores which are still unresolved. This technique is often referred to as *Speculative Store Bypass* (SSB) [1].

The objective is to assume that a load does not depend on a pending store unless there is a strong, visible reason to believe otherwise, hiding the latency of address generation. However, this optimization is responsible for opening a specific speculative window: if the prediction is incorrect, the load could transiently consume stale data from the memory hierarchy rather than the correct, value forwarded from the store buffer.

Microarchitectural structures (LSU, LQ, SQ)

The management of these out-of-order memory accesses is handled by the Load-Store Unit (LSU), which utilizes Load Queues (LQ) and Store Queues (SQ) to track in-flight operations. Speculative stores are located in a Store Buffer until they are architecturally committed, this creates a pool of pending data that will then be checked by younger loads. To maintain correctness intact without stalling, the LSU tries Store-to-Load Forwarding, where data from a buffered store gets passed directly to a dependent load if their addresses match. Since blindly stalling loads or searching buffers is very inefficient, dynamic predictors such as "Store sets" are often used [10].

These mechanisms track the history of conflicting memory operations to predict specific dependencies and force ordering only between loads and stores that have historically created problems with one another while allowing non-conflicting operations to proceed in parallel fashion.

Recovery and rollback

The correctness of these speculative reorderings usually is verified when the older store resolves its address or when instructions are preparing to retire. The hardware compares the address of the store with younger loads that have already executed, if a load is found to have read from the wrong location (thus, ignoring a dependency), a declaration of memory order violation occurs. After the detection is stated, the processor must flush the pipeline, squashing the faulting load and all following instructions in order to restore a valid architectural state. This squash event is crucial, because it will mark the end of the transient execution window [28].

Attacks like Spectre-V4 exploit this exact sequence, manipulating the predictor in order to allow a load to read sensitive data during the window and before the memory order violation forces the rollback [1].

2.2. Transient Execution Attacks (TEAs)

Given the microarchitectural foundations introduced in Section 2.1, we now describe Transient Execution Attacks (TEAs) as the composition of two fundamental mechanisms: a speculation trigger that opens a transient window and a microarchitectural encoding and recovery channel that persists beyond architectural rollback. This section first discusses microarchitectural state as a side channel. It then formalizes the interaction between

speculation and leakage, establishing a generic model that captures a broad spectrum of attacks. Finally, it focuses on the three attack classes targeted by the proposed framework and used in the experimental evaluation.

2.2.1. Microarchitectural state as a side channel

While the previous section detailed how processors execute instructions speculatively, the existence of transient instructions in itself does not constitute a security threat. For an attacker to exploit them, there must be a bridge between the transient domain, where architectural results are discarded, and the permanent domain. This bridge is formed by microarchitectural side channels.

Definition and context

We can define a *side channel* as an unintended information leakage mechanism in which secrets are possibly inferred from observable effects of an implementation, rather than from the system's intended nominal outputs [29]. These channels usually exploit shared resources, where the activity of one process can have an impact on the state of a resource observable by another. While traditional side-channel analysis often focuses on physical properties like power consumption or electromagnetic emissions, the most used one in the context of TEAs is timing analysis.

It is useful to distinguish between a side channel and a covert channel. A side channel usually implies that there is a victim inadvertently leaking data to an observer. But in attacks like Spectre, the transient gadget effectively acts as the sender in a *covert channel* which is internal to the process: the speculative code gets manipulated in order to encode intentionally a secret in the microarchitectural state [1]. This distinction is very important because it highlights that the "sender" (the sequence of transient instructions) and the "receiver" (the architectural recovery code) can be the same thread. This persistence of state where cache lines remain allocated or branch predictors remain trained also after a pipeline flush, violates the previously mentioned abstraction of the ISA, which tells that after a rollback discarded instructions should not leave any trace.

The cache as a side channel

The CPU cache hierarchy is the primary way for these channels to emerge, due to its being pervasive and its high signal-to-noise ratio. Modern processors use a hierarchy of caches (L1, L2, LLC) to hide memory latency. The important property exploited here is

the timing difference which is present between a cache hit (data served from SRAM) and a cache miss (data fetched from slow DRAM). Given that caches are finite and shared, the execution of one instruction stream can evict the data of another stream, or can also load data that a subsequent instruction can access quickly. To "probe" this state effectively, attackers use specific recovery primitives, we discuss the three most relevant to TEAs below:

- **Flush + Reload**: this primitive is the most precise, it offers cache-line granularity along with very low noise [44], it relies on the presence of shared physical memory between the attacker and the victim, the attack proceeds in three distinct steps

  1. Flush: the attacker uses a dedicated instruction, such as `clflush` on x86, to evict a specific memory line from the entire cache hierarchy
  2. Wait: the victim is allowed to execute (or is speculatively triggered), if the victim accesses the monitored line, the processor reloads it into the cache
  3. Reload: the attacker measures the time required to read that specific memory line, a fast read indicates the line was reloaded (a hit), implicitly telling that the victim touched the data, while a slow read indicates it remained evicted

  This technique is powerful especially because it targets a specific virtual address, avoiding the need to understand complex cache mapping policies.

- **Prime + Probe**: the attacker relies on this technique when shared memory is not available, it targets the cache sets rather than specific lines and exploits the limited associativity of the cache [22][38], once again it is split into three steps

  1. Prime: the attacker "primes" a specific cache set by filling all its ways with their own data
  2. Wait: the victim then executes, if the victim's working set maps to the same cache set, the victim's data will evict some of the attacker's cache lines
  3. Probe: the attacker measures the time to access their previously primed data, a significant delay in accessing any of the primed lines signals that they were evicted, and so, that the victim was active in that specific set [5]

  While it is more versatile than Flush+Reload since it does not require shared pages, Prime+Probe is noisier and also requires precise knowledge of the virtual-to-physical address mapping and the cache set indexing function to construct eviction sets which can be effective.

- **Evict + Time**: this technique can be seen as the inverse of Prime+Probe, it is generally used when the attacker cannot directly measure the victim's cache state but it can measure the victim's overall execution time [38]. The attacker evicts a specific cache set and then triggers the victim's execution, if the victim accesses data in that evicted set, it will generate a cache miss penalty, causing the overall execution to be slower.
  Even if it is less granular and often requiring many repetitions to filter noise, it serves as a viable alternative when other techniques cannot be used.

TLB as a side channel

While data caches are the preferred method for TEAs, other microarchitectural structures can be turned into side channels as well. The *Translation Lookaside Buffer* (TLB) is a specialized cache used to store recently used Virtual Address (VA) to Physical Address (PA) translations. Similar to the data cache, the TLB is a shared resource with limited capacity. A TLB hit avoids the very high latency penalty which would be given by a "page walk", where the Memory Management Unit (MMU) must traverse the page tables of the main memory [15].

Recent research, such as TLBleed, has shown that the TLB can be used as a reliable side channel by looking at the latency of memory accesses to infer the occupancy of a TLB set. The main difference is, since the TLB is distinct from the data cache, it can persist as a valid channel even when defenses such as cache partitioning (e.g. Intel CAT) are deployed to mitigate L3 cache attacks [22].

While the Proofs-of-Concept (PoCs) chosen and used in this thesis primarily utilize cache-based techniques for simplicity and reliability, it is important to acknowledge that the TLB can be utilized as well, and represents a viable microarchitectural channel for TEAs.

Encoding and recovery

The combination of transient execution and side channels establishes the generic operational pattern of TEAs, consisting of an Encoding phase and a Recovery phase.

1. **Encoding step (transient gadget)**: during the transient execution window, the attacker tricks the processor into executing a sequence of instructions (a gadget) that accesses a secret. This secret is then used as an index for a following memory access, usually into an array controlled by the attacker (e.g. probe_array), this

access takes the form of `probe_array[secret * stride]`, this operation loads a specific cache line determined by the secret value, a stride (often 4KB) is used to make sure that different secret values map to distinct cache sets in order to avoid interference with the hardware prefetcher [30].

2. **Recovery step (attack measurement)**: after the speculative window closes and the architectural state is rolled back by the processor, the side effects of the encoding step persist and the attacker then uses a primitive like Flush+Reload to iterate through the `probe_array`, measuring access times. The location of the "hot" (cached) line reveals the value of the secret [29].

This pattern highlights the critical property common to all TEAs: even though the processor discards the architectural results of the transient execution, the microarchitectural state changes persist and are measurable, this generic model of encoding and recovery is the foundation for the modeling of these attacks, which will be presented in the following section.

2.2.2. TEA model

We define a generalized execution model for TEAs to provide a template to use in order to analyze specific attacks such as Spectre (Section 2.2.4), Spectre-v4 (Section 2.2.5), and Meltdown (Section 2.2.6). A successful TEA can be decomposed into the following five stages:

1. **Trigger primitive**: the attack starts by inducing a microarchitectural event which causes the processor to execute instructions that will be discarded afterwards. This is obtained via mechanisms like mispredicting a conditional or indirect branch (control speculation), mispredicting a memory dependency (data speculation) or raising a delayed exception (fault-based execution).

2. **Transient window**: after the trigger, the processor is in a state of transient execution, during this period, instructions are fetched, decoded and dispatched to execution units, while their results are not committed to the architectural state, they are valid operands for dependent micro-operations within the pipeline.

3. **Secret access**: during the transient window, the speculative instruction sequence performs an operation that tries to access sensitive data, usually a memory load from a kernel address (in Meltdown) or an out-of-bounds array read (in Spectre). Given that permission checks and bounds checks are often resolved at instruction retirement or after the processor verifies the speculation, the access using the secret

data is able to occur.

4. **Encoding**: to be able to exfiltrate the secret data before the squash of the instructions, the transient sequence must convert the data into a persistent modification of the microarchitectural state. The standard technique involves using the secret value as an index for a memory access to a shared data structure (e.g. a "probe array"). This operation brings specific cache lines into the hierarchy or updates TLB entries, creating a footprint that survives the flush of the pipeline.
5. **Recovery**: after the processor detects the mis-speculation or fault, it discards the transient instructions and reverts back the register state, the attacker then executes the recovery, using side-channel primitives such as Flush+Reload or Prime+Probe (described in Section 2.2.1) to measure the state of the used shared resources. By identifying which cache line or set shows lower latency in terms of access, the attacker can reconstruct the secret value used during the encoding stage.

2.2.3. Transient windows in practice

The success of a TEA is strictly tied to the transient window, this section details the microarchitectural origins of these windows and the factors that modulate their duration.

Definition and boundaries

The transient window starts when the processor issues an instruction based on prediction (control or data) or executes an instruction which will eventually fault. The window ends when the mis-speculation is cleared (correct path is chosen) or the exception is handled (the faulting instruction reaches the head of the Reorder Buffer). At this point, the pipeline gets flushed, removing all younger instructions from the ROB and reservation stations. The duration of the window determines how many dependent operations (especially the secret access and encoding) can be executed before the rollback happens [1].

Three major window sources

We can classify transient windows based on the trigger primitive that starts them, these are correlated directly to the specific attack variants which will be discussed later.

Control speculation windows: these arise from the branch predictor, when a branch instruction is encountered, the processor predicts the direction (taken/not-taken) or the

corresponding target address, the window is persisted until the branch execution unit resolves the actual outcome and compares it with the prediction. If a mismatch happens, the window closes. As discussed in Section 2.1.3, complex pipelines and predictors (like BTB and RSB) are essential for performance but create substantial windows for exploitation if mistrained [1][20].

Data speculation windows: these are generated by memory disambiguation logic, since, as stated in Section 2.1.4, to maximize parallelism, processors may speculatively issue a load instruction before older store addresses are fully resolved (Speculative Store Bypass). The window exists while the core thinks there is independence between the older store and the younger load, it ends when the store address is computed and the Load-Store Unit detects a memory order violation, triggering a squash of the instruction [1].

Exception/Fault-driven windows: these windows occur when an instruction raises an exception (e.g. a page fault), in Out-of-Order processors, exceptions are typically handled at instruction retirement (Section 2.1.2). As a consequence, instructions following the faulting instruction could execute transiently until the faulting instruction reaches the commit stage, at that point the pipeline is flushed, this mechanism is used by attacks like Meltdown [30].

Window duration factors

The "depth" of a transient window, which is defined by how much work can be accomplished within it, is governed by several microarchitectural factors which will be described now.

Resolution latency: the primary constraint is the time required to solve the trigger condition, for a Spectre attack for example, this is the time to evaluate the branch condition. If the operands for the comparison are not in the cache (e.g. a cache miss on the array length variable), the resolution can take hundreds of cycles, providing a very large window for transient execution.

Instruction window limits: the capacity of the Reorder Buffer (ROB), issue queues and load-store queues imposes a hard limit on the number of transient instructions that can be actually executed, modern microarchitectures often support hundreds of active instructions, possibly offering space for complex gadgets [41].

Memory latency and outstanding misses: chained cache misses can artificially extend the window, if the trigger instruction depends on a long-latency memory operation, the processor will try with aggressiveness to fill the pipeline with speculative work in order to hide this latency.

Pipeline width: wider superscalar architectures that issue more micro-operations per cycle allow a larger transient footprint for each unit of time, potentially allowing complex gadgets to complete even with relatively short temporal windows.

Practical implications for TEAs

The dynamic nature of the transient window allows the feasibility of specific attacks. Attackers prefer gadgets that are computationally inexpensive and have few dependencies to ensure that the encoding step completes before the window closes. A longer window (e.g. caused by an L3 cache miss on branch condition) allows for richer gadgets, like traversing linked lists or performing complex arithmetic before encoding. On the contrary, shorter windows restrict the attacker to simple array accesses but are still exploitable if the encoding to the microarchitectural state is sufficiently fast.

2.2.4. Spectre V1 (Bounds Check Bypass)

Spectre Variant 1 (Bounds Check Bypass) represents the canonical instantiation of the TEA model described in Section 2.2.2, which uses a control speculation trigger, it exploits the conditional branch predictor (Pattern History Table or PHT) to execute architecturally forbidden memory accesses while inside a transient window.

Trigger

The attack is started by mistraining the CPU's branch predictor, modern processors maintain a history of branch outcomes to predict future execution paths. The attacker invokes a specific conditional branch, typically an array bounds check such as `if (x < array1_size)`, multiple times with valid inputs. This as a result trains the PHT to predict the branch as "taken" (true), but once the predictor is trained, the attacker uses a malicious input `x` that is out-of-bounds. Because of the latency involved in resolving the comparison (e.g., if `array1_size` is not in the cache), the processor enters a speculative state, predicting the branch as taken and opening the transient window, during this window the instructions in the branch body get executed speculatively using the malicious

index.

Transient gadget

The core of the Spectre V1 gadget involves a sequence of dependent instructions that transform an out-of-bounds read into a microarchitectural side effect, the standard pattern is described by Kocher et al. [29] using the C language:

Listing 2.1: Spectre V1 gadget

```
1 if (x < array1_size) {
2     y = array2[array1[x] * 256];
3 }
```

In this sequence, array1[x] performs the out-of-bounds read, retrieving a secret byte k from the victim's memory space, this secret value is then used to calculate the address for a following memory access into array2 (probe array). This creates a data dependency because the cache line loaded from array2 depends entirely on the value of the secret k. To ensure the encoding occurs before the discarding of the pipeline, the gadget must have a minimal dependency chain and avoid stalling conditions other than the initial bounds check resolution [29].

Once the bounds check is resolved and the processor notices the misprediction, the architectural state is rolled back, but the cache line corresponding to `array2[k * 4096]` remains present in the cache hierarchy. As mentioned in Section 2.2.1, the attacker typically employs Flush+Reload (used in our chosen Proof-of-Concept) or Prime+Probe to measure access times to the probe array, identifying the cached line and recovering the secret byte `k`.

Practical considerations

The prevalence of Spectre V1 vulnerabilities comes from the presence of array bounds checks everywhere in software, especially in compilers, interpreters and operating system kernels. Furthermore, the successful exploitation often requires manipulation of the cache state (e.g. flushing `array1_size`) to increase the length of the transient window sufficiently to complete the encoding step.

2.2.5. Spectre V4 (Speculative Store Bypass)

Spectre Variant 4, also known as Speculative Store Bypass (SSB), instantiates the generic TEA model using a memory-dependence speculation trigger.

Trigger

In Out-of-Order processors, loads can execute speculatively before the addresses of all older uncommitted stores are known, the memory disambiguator predicts whether a younger load overlaps with an older unresolved store. If the predictor assumes "no conflict" the load executes immediately, retrieving data from the memory hierarchy (L1 cache) instead of waiting for the store to pass the correct data from the Store Buffer. If the addresses actually overlap, the load transiently uses stale data, the transient window exists until the older store resolves its address, a dependency conflict is noted, and the pipeline is flushed [1].

Transient gadget

A typical V4 gadget involves a sequence where we have a store writing a safe value to a sensitive stack location, followed then by a load from that same location, at that point if the store's address calculation is slow (e.g. dependent on a cache miss), the load could speculatively bypass the store and read the "stale" value present in memory before the store happened. If the stale value is a sensitive pointer or a secret, it can be passed to a dependent operation (e.g. `probe_array[stale_value * stride]`) to encode it into the cache.

Upon detection of the violation of the memory order, the pipeline gets flushed, but the cache modifications remain and can then be recovered using one of the techniques mentioned in Section 2.2.1.

Conceptual difference from V1

Spectre V4 differs conceptually from V1 in the nature of the speculation:

- Spectre V1 (Control Flow): the processor executes instructions on the wrong path due to a branch misprediction.
- Spectre V4 (Data Flow): the processor executes instructions on the correct path (architecturally valid) but these instructions transiently use the wrong value (stale data) because of incorrect memory disambiguation.

2.2.6. Meltdown-US (Supervisor-only Bypass)

Meltdown (specifically Meltdown-US) instantiates the TEA model using a transient window which is exception-driven, it exploits a race condition between the execution of a memory load and the architectural handling of the privilege violation exception.

Trigger

The trigger for Meltdown is a load instruction that issues a memory request to a supervisor-only (kernel) address from user space, in Out-of-Order microarchitectures, instructions are fetched, decoded into (μOPs) and executed as soon as their operands are available. It is very important to note that the exception resulting from the privilege violation is typically not handled immediately upon execution, instead, the exception is raised only when the instruction tries to retire (commit to architectural state). This creates a transient window between the execution of the load and the pipeline flush [30].

The core vulnerability can be found in the separation of the data fetch mechanism from the permission check logic. To minimize latency, the processor speculatively fetches the data from the L1 cache and sends it to dependent operations in parallel with the privilege check. Even though the permission check fails, preventing the register state from being updated architecturally, the data has already been sent to subsequent (μOPs) in the pipeline. This allows the transient instructions to compute on data which is architecturally forbidden [29][30].

Gadget

The canonical Meltdown gadget is very simple compared to Spectre, because it does not require complex operations to be performed like mistraining. The standard pattern is reported below, as described by Lipp et al. [30] using the Assembler language with Intel syntax:

Listing 2.2: Meltdown gadget

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

In the above snippet of code an inaccessible kernel address is moved to a register, raising an exception, then, following instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access [30].

Prerequisites

It is critical to note that practical exploitation of Meltdown-US is based on specific microarchitectural behaviors present in Intel CPUs (and some ARM variants) prior to hardware fixes like Rogue Data Cache Load (RDCL) protection. It is also important to say that operating system mitigations such as Kernel Page Table Isolation (KPTI/KAISER) effectively mitigate the attack by unmapping kernel memory from the user address space. As a consequence, the experiments discussed in this thesis assume either the use of older hardware or the intentional disabling of KPTI to demonstrate the vulnerability mechanisms [1].

2.3. Threat model

2.3.1. Scenario description

We consider an Amazon-like multi-tenant compute platform where customers submit code that is executed on shared physical servers. Tenants are isolated using a strong virtualization boundary (microVMs or similar sandboxes), so we assume that architectural isolation holds, meaning that one tenant cannot read another tenant's memory by exploiting standard software vulnerabilities alone. At the same time, the platform is performance-driven, so microarchitectural resources are shared across tenants on the same host (last-level cache, memory subsystem, TLBs, branch predictors and possibly core-private structures under certain scheduling conditions). This is the setting where TEAs are relevant, even if memory is isolated at the page-table level, transient execution can still create measurable footprints that cross boundaries of the tenants.

In this scenario, the provider controls the build/deployment pipeline for workloads of each tenant, the proposed framework is deployed as a compile-time hardening stage since it scans the code for patterns which are prone to transient execution, rewrites the vulnerable regions into hardened variants and selects a configuration that satisfies a chosen security/overhead objective. The binary obtained from the Assembler file outputted by the framework is what is actually deployed in the platform. The victim is a benign ten-

ant workload that we assume handles sensitive data (keys, tokens, proprietary payloads, model weights) while the attacker is another tenant trying to obtain co-residency on the same host to steal those secrets through microarchitectural side effects.

Finally, we explicitly assume a realistic platform baseline: Meltdown-class issues are typically mitigated in production through OS-level page-table isolation and vendor updates. For completeness and comparability with prior work, Meltdown-style PoCs are still useful as an experimental reference point, but the primary cloud-relevant focus is on Spectre-style transient behaviors that remain meaningful even under modern system hardening.

2.3.2. Attacker capabilities

The attacker is modeled as an unprivileged tenant who can submit and repeatedly run arbitrary code inside their own isolated execution environment (microVM/container) with no kernel privileges. They could become co-resident with the victim on the same physical host because of normal multi-tenant scheduling and they can repeat measurements across many runs. We make a conservative assumption on scheduling: co-residency can occasionally place the attacker and the victim on the same package, let them share the LLC and the platform may enable Simultaneous Multi Threading (SMT), which increases sharing of core resources and generally increases the attacker's ability to observe microarchitectural effects.

We also assume the attacker has enough measurement capability to form cache-based side channels without relying on special sharing assumptions. In practice, we do not require shared pages between attacker and victim (so we do not have to rely on Flush+Reload as a prerequisite in the deployment model). Instead, the attacker can use cross-domain techniques like Prime+Probe against shared caches, together with timing sources available to normal user processes (even if rdtsc/rdtscp are restricted, other timing interfaces are often enough). For Spectre V1 and V4, the attacker is assumed to have an interaction channel that influences the victim's inputs, which is very realistic in cloud systems where workloads expose APIs or shared services, this input influence is what makes training possible along with the repeated triggering needed for TEAs.

Additionally, the attacker does not control the victim's binary or compilation process, it cannot disable platform mitigations and cannot directly read victim memory through architectural means. Their advantage comes only from repeated execution, co-residency and microarchitectural observation. This is exactly the class of threat that needs code

hardening against TEAs in multi-tenant environments.

2.3.3. Assets and security goals

The primary asset in this scenario is tenant confidentiality, meaning that secrets that are processed by a benign workload (e.g. cryptographic keys, authentication tokens, proprietary input data) should not become learnable by a co-resident attacker through transient-execution side effects. Furthermore, isolation is extremely meaningful, even if an attacker can run arbitrary code, they should not be able to let microarchitectural side effects become a reliable way to read other tenants' data.

Given the scope of the framework, the security goal is not to eliminate speculation, but instead is to prevent or substantially degrade the ability of transient execution to create a reliable secret-dependent encoder. In TEA terms, the framework targets the secret-dependent access and encoding stages by rewriting vulnerable patterns so that, if a misspeculation happens, secrets cannot be used as an address/index (or can only do so in a masked non-informative way) and by providing configurations that reduce the repeatability of the leakage. So the goal in the end is to raise the attacker's cost and reduce reliability for the extraction of the secret bytes in the multi-tenant setting.

2.3.4. Assumptions and non-goals

We assume that the cloud provider imposes strong architectural isolation (microVM/-container boundaries) and that the provider controls the compilation and deployment pipeline for the protected workloads, so the framework's rewritten output is the Assembler file that is actually compiled and executed. Co-residency is possible but not guaranteed and we conservatively assume shared last-level cache resources and potentially SMT, since these increase the feasibility of microarchitectural observation. At the same time, we do not assume any special sharing conditions (e.g. shared pages between tenants) and the threat model does not depend on privileged access or disabling platform mitigations. For Meltdown specifically, we assume a modern production baseline where OS/hardware mitigations are present, Meltdown-style evaluation is in fact used primarily as a reference point rather than as the main threat for the cloud platform.

Non-goals follow naturally from these assumptions, the framework does not aim to:

- mitigate every TEA variant, especially those that are primarily BTI/V2-driven or require hardware defenses

- eliminate cache side channels in general (Prime+Probe, Flush+Reload, timing channels and similar mechanisms still exist as measurement techniques)
- guarantee constant-time behavior for arbitrary application code

The focus is on code-level hardening of detectable transient-execution patterns and on selecting mitigations that are deployable and tunable in a production pipeline, instead of being on replacing OS-level isolation or microarchitectural channel defenses.

2.3.5. Success criteria

Success is defined by measurable improvements in leakage resistance and overhead in a controlled experimental setup that mirrors key constraints of multi-tenant deployment. From a security perspective, the hardened binaries should exhibit a clear reduction in transient leakage, in the sense that TEA Proof of Concepts for the targeted variants (Spectre V1/V4-style patterns and Meltdown) become less reliable, requiring significantly more repetitions to extract the same information or fail to recover secrets. From a performance perspective, the framework should demonstrate that it can avoid the worst case behavior of simple mitigations (e.g. fencing everywhere) by applying rewrites to specific targets and by selecting lower-overhead variants where possible.

In practice, we consider the framework successful if it can meet the following qualitative targets simultaneously:

- it substantially degrades leakage outcomes in representative PoCs (lower success rate, higher error or higher complexity for the attacker).
- it keeps overhead within a reasonable budget and shows that different security–performance operating points can be chosen depending on policy.
- it is compatible with an automated build-and-deploy pipeline, although the current prototype is not integrated into LLVM and would require further engineering work for production deployment.

3 | Related works

This chapter outlines software defenses against TEAs while organizing them by methodological families (barrier-based hardening, data-flow / dependency hardening, analysis-driven selective rewriting, microarchitectural footprint confusion and Meltdown mitigations like OS-level isolation).

For each family, we discuss representative tools and proposals in depth, focusing on how they work, which assumptions they are based on, their strengths and their main limitations. The chapter closes by positioning this thesis within the state of the art by making explicit which gap the proposed framework is meant to address.

3.1. Scope and selection criteria

This review focuses on software-oriented defenses against TEAs that match the threat model and objectives of this thesis. In particular, it targets Spectre V1 (bounds-check bypass), Spectre V4 (speculative store bypass, SSB), and Meltdown (Meltdown-US), discussed in Sections 2.2.4, 2.2.5, and 2.2.6. While these attacks differ in how speculation is triggered, they can be described through a common pipeline (Section 2.2.2). By using it we are able to compare defenses based on where they intervene, rather than treating each variant in isolation.

To keep the discussion focused, we exclude several classes of work. We omit Spectre V2 and branch target injection defenses (e.g. retpoline, IBRS/STIBP, BTB/RSB techniques), since they primarily address indirect control-flow speculation mechanisms that are not central to the V1/V4 patterns considered here. We do not consider hardware-only proposals that require new CPU features or microarchitectural redesign because they fall outside a software-centric study. Finally, attack-only papers are not reviewed unless they introduce a defensive mechanism.

Within this scope, the works discussed in this chapter are selected to be representative of the main software mitigation directions. In practice, a work is included if it mitigates

at least one of the target variants, provides a well-specified mechanism (ideally with a prototype), and reports quantitative results that allow reasoning about both protection and overhead.

For consistency, each work is evaluated based on: coverage (which patterns and contexts are protected and what remains out of scope), the security argument and its dependence on attacker-model assumptions, performance overhead, and deployability. The remainder of this chapter uses these criteria to group the literature into methodological families and to position the framework proposed in this thesis.

Note that some works naturally span multiple families, for example, oo7 [42] combines barrier-based hardening with program-analysis-driven selective rewriting.

3.2. Barrier-based hardening

Barrier-based hardening is the most direct software mitigation family for transient execution attacks. The idea is to constrain speculation right at the boundary where the transient behavior would allow sensitive data to affect the microarchitectural state. This is done by inserting primitives which stop speculation (typically fence-like instructions) so that execution cannot go on past a guard until the relevant condition is resolved architecturally. For Spectre V1, barriers are placed around bounds checks or other control-dependent guards to prevent that the "unchecked" path perform transiently secret-dependent loads or indexing. For Spectre V4 (Speculative Store Bypass), the same principle is applied to memory-ordering: fences are used to prevent a load from transiently seeing stale data when an older store has not been fully resolved yet.

The clear advantages of this family are its simplicity and relatively local nature, it can often be deployed without complex global analysis and is therefore commonly used as a baseline or a rapid mitigation. The downside is the high cost, because barriers reduce instruction-level parallelism and reduce drastically the performance benefits of speculation, especially when applied conservatively. As a result, the main design problem is their placement since coarse-grained fencing is robust but expensive, while fine-grained fencing wants to reduce overhead by restricting barriers only to the specific checks and uses that can participate in transient secret encoding but may end up reducing the protection.

3.2.1. VeriFence

VeriFence [19] is very close to classic barrier-based hardening, but it includes a smart policy for where fences should go. The paper focuses on unprivileged eBPF (extended Berkeley Packet Filter) programs in Linux, where the kernel verifier must already reason about safety by itself and, after Spectre the verifier was extended to consider speculative paths too. The problem is that these Spectre-aware checks can become pessimistic and reject a large portion of real programs, which in practice pushes users to disable defenses. VeriFence tries to change the strategy, it wants to verify speculative paths optimistically and when it encounters a speculative path that would be unsafe (like type confusion or out-of-bounds), it falls back to inserting a speculation barrier so that the transient path is terminated rather than rejecting the whole program. In other words, the fence becomes a targeted "cut" in the speculative control-flow graph, guided by the verifier evidence, and the result is that previously rejected programs can be accepted while still preserving the kernel's safety goals.

VeriFence is pragmatic, it builds on the existing eBPF verifier and uses fences as a targeted fallback only when speculative paths would violate safety invariants, so it improves acceptance of real-world programs without defaulting to blanket fencing. In that sense it achieves a nicer security–overhead compromise and fits well the kernel/BPF workflow [19].

It inherits assumptions from the verifier's model since it relies on the idea that the inserted barrier actually stops speculation in the way the defense expects and it also reasons under a leakage model that is conservative but still a model, not a full characterization of every possible microarchitectural channel. Also, it is scoped to protecting the kernel from untrusted BPF code, not to providing general constant-time style protection for arbitrary application logic, so the portability of the idea outside that setting is limited [19].

3.3. Data-flow / dependency hardening

Data-flow (or dependency) hardening is a mitigation family that tries to "keep speculation on", while trying to make it unhelpful for leaking secrets. Instead of stopping the CPU with a fence, the code gets rewritten so that the values that could become dangerous under misspeculation (typically an unchecked index or a secret loaded transiently) are either delayed by an artificial dependency or forced through a mask/poisoning step. The goal is simple because even if the branch predictor gets the path wrong, the transient window should not be able to turn attacker-influenced state into a secret-dependent address, index

or other microarchitectural effect that survives the squashing.

Conceptually, these techniques sit between two extremes. On one side there is full serialization (e.g. putting `lfence` after every conditional branch), which is robust but tends to destroy ILP, on the other side there is attempt of "doing nothing" and hoping that gadgets are rare, which is not acceptable. Dependency hardening instead uses the CPU's own ordering rules: if a load's inputs depend on the condition flags or on some value that cannot be ready before the check resolves, then the load cannot execute early (or it executes but gets neutered via masking). It is still not free, but the overhead can be smaller than fencing everything, at least for many workloads.

3.3.1. You shall not bypass

The You shall not bypass [37] report starts from the practical observation that Spectre V1 (Bounds Check Bypass) is hard to "patch away" at the system level and naïve fencing is too expensive if applied everywhere. The paper explores a set of patterns that introduce artificial data dependencies between the outcome of a bounds check (i.e. branch condition) and the memory operations that would otherwise form the leaking gadget. They discuss multiple ways to realize this: reading EFLAGS via `LAHF` and mixing it into a reserved register to create a dependency chain, using conditional moves in the style of Speculative Load Hardening (SLH) to derive a mask, and a weaker variant that depends only on the comparison arguments. The central point is that this techniques are able to delay the dangerous loads (or make their values masked) without globally serializing the entire basic block. In their evaluation on Phoenix benchmarks, they show that "`lfence` after every conditional branch" can be extreme (hundreds of percent slowdown), while dependency-based approaches are much cheaper on average, roughly on the order of tens of percent overhead.

The work is very valuable because it frames the problem as "delay only what matters", rather than turning off speculation everywhere. It also breaks down the design space in a very concrete way (what to depend on, which ISA tricks to use, what register pressure gets introduced) and the performance results conclude that dependency hardening can be a realistic baseline when fences are not [37].

The main limitation is that the guarantees depend on microarchitectural assumptions, some variants are explicitly acknowledged as weaker (e.g. depending only on comparison arguments does not strictly enforce ordering) and future CPU features like value predic-

tion could hurt “dependency-only” ordering unless you also mask values. There is also a non-trivial engineering cost since a a reserved register is often needed and also extra instructions on hot paths [37].

3.3.2. Ultimate SLH

Ultimate SLH [2] revisits SLH not as a single technique but as a family of compiler-inserted transformations with different security and performance properties. The authors compare the LLVM implementations that poison loaded values (vSLH) or poison load addresses (aSLH) and they also implement “strong SLH” (SSLH) as proposed in prior work, which expands poisoning to more sources of leakage (e.g. stores and branch conditions). A key contribution of this work is showing that even these stronger variants still fail to stop all Spectre V1 attacks in practice, because variable-time arithmetic instructions can leak information even if executed only transiently. From there they propose “Ultimate SLH” (USLH), which extends SSLH by additionally poisoning inputs to variable-time arithmetic instructions (and handling repeat instructions), with the goal of aligning speculative leakage with what constant-time programming already considers unsafe. They implement USLH in LLVM and evaluate on SPEC2017, showing that it provides broader protection than prior SLH variants while remaining consistently faster than an lfence-style approach that blocks all speculation at branches.

The paper has a strong methodology because it does not just propose a tweak, it systematically checks what SLH variants actually cover, then demonstrates a concrete missing channel and in conclusion patches the gap with an implementation we can reason about in compiler terms. The connection they make to constant-time requirements is also useful, because it gives a clean conceptual bridge between Spectre safety and existing side-channel discipline [2].

USLH is still a heavy compiler transformation, it needs speculation tracking, extensive poisoning and this can create real overhead and register pressure (some benchmarks slow down a lot even if it beats fencing). Also, the work is focused on Spectre V1 and its leakage model, it improves coverage inside that box, but that doesn’t automatically generalize to every transient mechanism or every future covert channel and variants. Furthermore, in some corner cases (e.g. certain floating-point paths) the approach still goes back to more simple tools like fences [2].

3.4. Program analysis (selective rewriting)

Program-analysis-driven selective rewriting sits in between "blindly harden everything" and "trust developers to hand-patch gadgets". Its idea is to first identify, with static or hybrid analyses, the specific code regions where speculative execution can turn into a transient secret-dependent access and then apply a targeted patching strategy only there. In practice this usually means building some notion of speculative path feasibility (even if approximate), tracking attacker-controlled inputs via taint/data-flow, and mapping them to "sinks" [35] like memory address computations, array indices, or control-flow decisions.

What makes this family attractive for related work is the explicit detector to patch pipeline: we get a set of reported vulnerable fragments and a concrete rewriting action (insert a fence, rewrite a pattern, etc). The big trade-off is that the security depends on the analysis model (e.g. how speculation windows are bounded, what speculation primitives are assumed), while the performance depends on how selective the rewriting really stays once we move from toy gadgets to real codebases.

3.4.1. oo7

oo7 [42] proposes a static, binary-level workflow that detects Spectre-vulnerable fragments and then patches the binary by inserting fences only where needed. The detection side combines control-flow extraction, taint analysis and address analysis. The tool targets Spectre Variant 1 and closely related patterns (including the write-oriented variants 1.1/1.2), by identifying tainted conditional branches and the subsequent speculative reads/writes within a speculative execution window, then placing `lfence` at selected points rather than after every branch. The paper reports that oo7 detects all fifteen purpose-built Spectre litmus patterns and uses selective `lfence` insertion to block the demonstrated leaks.

The main upside is deployability, the approach operates at the binary level and does not require hardware or OS changes, so in principle it can be used as a patching step for existing executables. It is also purposely selective, because fences are inserted only for branches classified as tainted and on the concrete reconstructed vulnerable sequences, which is exactly how it keeps overhead low compared to techniques that insert fences everywhere. The evaluation mixes litmus-style functional validation with SPECint performance and a study over many binaries [42].

The security coverage is strongly tied to the analysis precision because for instance, the paper explicitly notes that handling Spectre Variant 4 would require a more accurate address analysis than what they currently support, so the method does not cleanly generalize to SSB-style patterns in the presented form. It also relies on assumptions about the speculative window size and the paper's own discussion suggests that scaling analysis already costs much time on real programs, this hints at a practical limit for "deep" modeling of speculation while staying usable. Finally, it intentionally does not target V2/RSB-style issues, so it fits best as one component in a bigger mitigation method rather than a complete answer by itself [42].

3.4.2. SERBERUS

SERBERUS [35] is a compile-time hardening approach aimed at constant-time cryptographic code, trying to make sure that code is robust even when transient execution can lead secrets into otherwise "public" operands. The paper's framing is that once you adopt a stricter discipline (they define a stronger notion around speculative constant-time and additional well-formedness constraints), then leakages can be characterized using a small set of taint primitives and mitigations can be applied systematically. Concretely, SERBERUS is implemented as LLVM passes that:

- construct a transient control-flow view and insert `lfence` in a loop-aware way to avoid destroying hot loops.
- introduce function-private stacks to reduce leakage paths tied to calling/return behavior.
- apply register cleaning so secrets do not remain in registers across boundaries where misprediction could expose them.

On their benchmarks (OpenSSL, Libsodium), the reported average overhead is 21.3%, and they argue this is lower than baseline stacks of mitigations they compare against, with fence insertion being the dominant cost driver.

Compared to many ad-hoc "patch gadgets" techniques, SERBERUS is more systematic as it ties the mitigation design to an explicit leakage characterization for the target domain (static constant-time crypto) and it integrates directly in the compiler pipeline. Performance-wise, the loop-aware fence placement matters a lot for crypto and the results show that avoiding fences inside tight loops can keep large-buffer throughput from collapsing. It is also broader than pure PHT-only hardening, since it is designed around multiple speculation primitives [35].

The price to pay is in assumptions and scope, SERBERUS is tailored to cryptographic code that already follows the static constant-time discipline, so it is not a general-purpose mitigation for arbitrary C/C++ programs. It also assumes a specific hardware "mode", which means some guarantees are conditional on platform features and configuration rather than actually software-enforced. Finally, the paper notes that fully covering certain behaviors (e.g., PSF in their discussion) is not something they can just "fix in software" without losing the low-overhead property, so there is an explicit boundary to how far this strategy can stretch [35].

3.5. Microarchitectural footprint confusion

Microarchitectural footprint confusion is a family of mitigations that does not try to solve speculation itself, instead it targets the observability of transient effects made possible by side channels. The core assumption is that many TEAs become practical only because the attacker can create a clean microarchitectural signal and measure it with enough timing precision to recover bits reliably. If we can inject noise, remove key measurement primitives or make the microarchitectural footprint less repeatable, the transient window can still happen, but the encoding/recovery phase becomes too unreliable to exploit in practice.

Basically these defenses work one step later in the TEA pipeline because they accept that secrets could transiently influence cache/TLB state, but they attempt to degrade the side channel capacity. This tends to be a pragmatic direction in OS settings because it can be deployed as a system policy (kernel, runtime, platform configuration) and it does not require rewriting every application, but of course the downside is that it rarely achieves a "no leakage" guarantee, its goal is more like trying to push attacks back into having high-noise issues.

3.5.1. FlushTime

FlushTime [18] proposes a kernel-level mechanism for ARMv8-A that targets a very common pattern in modern cache attacks: attackers often use cache flush instructions (ARM cache maintenance instructions, the same as x86 `clflush`) to reduce noise and then they rely on a high-resolution timer to distinguish cache hits from misses. The paper's main idea is to make these two primitives collaborate: cache flush instructions and accesses to the generic timer are trapped from user space into the kernel (using ARMv8-A trap con-

trols) and whenever a process executes a flush instruction the kernel temporarily reduces the time resolution returned by the generic timer for a short window measured in a number of context switches. The resolution reduction is implemented by masking low-order bits of the timer value, so just after a flush the attacker loses the fine-grained timing needed for Flush+Reload style recovery, while normal applications (which rarely need nanosecond timing right after a flush) are expected to keep working normally. The implementation is done by small Linux kernel modifications and the evaluation shows resistance against several flush-based attacks (including flush-based Spectre and Meltdown demonstrations on ARMv8-A) with low system-level overhead reported on UnixBench and SPECrate2017.

The design targets a concrete "enabler" of many practical attacks rather than focusing individual gadgets, so it generalizes across multiple flush-based techniques without needing application changes. It is also deployable on existing ARMv8-A systems since it relies on trap mechanisms that already exist and the paper argues that the performance impact is small at the system level [18].

The coverage is explicitly scoped as it is effective against flush-based cache attacks, but it does not stop attacks that avoid flush instructions (e.g., Prime+Probe style) or attackers that can obtain timing via alternative sources not covered by the mechanism, the paper discusses these gaps quite openly. Also if an attacker continuously flushes, the system can be pushed into extended low-resolution timing periods which starts to look like a denial-of-service on "good" timing and the defense becomes less transparent to benign workloads that really need high-resolution measurements. Finally, portability to other architectures or to different timer/flush configurations is not automatic, in fact it would require re-engineering [18].

3.6. Control-flow manipulation & diversification

Control-flow manipulation and diversification approaches start from a practical observation: many Spectre-style exploits are not only about reading a secret transiently but they are about reliably leading speculative control flow so that the CPU reaches a useful gadget again and again. If the attacker cannot predict where speculation will land, the exploitability drops sharply, so the mitigation acts on the control-flow surface, it rewrites how branches, calls/returns, and indirect jumps are expressed, and sometimes it also randomizes the placement of code to make repeatable gadget selection harder.

Diversification complements manipulation rather than replacing it, manipulation gives

structural safety properties (e.g. "speculation can only enter at safe block boundaries"), while diversification tries to break the attacker's ability to line up congruent predictors and stable offsets (especially in out-of-place attacks). In practice, these defenses tend to look like compiler and runtime engineering: code layout changes, new calling/return conventions and replacement of certain control-flow constructs with safer ones [8].

3.6.1. Swivel

Swivel [36] is a compiler framework that hardens WebAssembly (used as an in-process sandbox) against Spectre by combining control-flow reshaping with a tunable "security vs overhead" design. The core idea is to compile Wasm into linear blocks, i.e. straight-line code regions where every control transfer (including calls) happens only at block boundaries and where safety checks like memory masking are kept local so speculation cannot bypass a check performed somewhere else. After that Swivel hardens the control-flow edges: in the software-only design (Swivel-SFI) it avoids `ret` entirely by using a separate return stack and translating returns into controlled jumps, and it uses BTB flushing on sandbox boundary to limit cross-domain predictor reuse. For conditional-branch poisoning, the deterministic variant rewrites conditional branches into indirect jumps (CBP-BTB conversion) so it can rely on a predictor that can be flushed. Talking about diversification, Swivel offers an ASLR-based mode that randomizes sandbox code page placement (and combines it with BTB flushing at boundaries), which does not prove poisoning to be impossible but raises the bar and makes repeatability harder in the multi-tenant setting they target.

Swivel is a good representative of this family of mitigations because the mitigation is a coherent control-flow design that we can reason about at the CFG level, linear-block boundaries, constrained targets, and hardened call/return behavior. It also exposes two clear operating points, probabilistic (ASLR-based) and deterministic (elimination-oriented). However, the paper is quite explicit about practical deployment constraints in sandboxes and about performance as it shows that avoiding fencing everywhere can matter by an order of magnitude in some settings [36].

The most important limitation is scope and context because Swivel is strictly related to a sandboxed execution model (Wasm and a specific compiler/runtime pipeline), so we cannot automatically assume the same transformations transfer cleanly to native C/C++ binaries without losing the nice invariants. Also, the deterministic modes are costly on certain control-flow patterns (especially data-dependent loops), the paper itself shows that

overhead is not uniformly small and sometimes it becomes hard to justify. Furthermore, while the ASLR-based option looks good and pragmatic, it is still probabilistic and does not give a universal “no gadget reachable under speculation” statement [36].

3.7. Meltdown-oriented software mitigations

Mitigations for Meltdown-class attacks tend to look different from “classic" Spectre V1 solutions. The reason is that Meltdown is fundamentally about privilege checks happening too late, so the attacker can transiently read kernel-only data and then encode it through caches.

A pure software response therefore often has the goal to change how the OS maps kernel memory, how it separates privilege domains and when and how it switches page tables. In practice, this is the KPTI-style idea, keep the kernel mostly unmapped while running user code, then map it only on kernel entry, even if that makes transitions heavier.

A second line of work tries to make this mapping discipline less expensive and more structured, instead of a single global kernel mapping, the kernel is reorganized so that common syscalls can run with a limited, process-specific "safe” kernel view and only switch to a fully privileged view when it really needs to touch sensitive global state. So the mitigation becomes architectural at OS level: memory layout, per-process views and careful partitioning of kernel data structures.

3.7.1. KAISER / KPTI (Kernel Address Isolation)

KAISER’s [23] main idea is straightforward but powerful, in user mode, do not keep the full kernel virtual address space mapped, because “mapped but inaccessible” still leaks information via microarchitectural effects. The design splits the page tables, so user processes run with a page table that maps user space plus only a tiny kernel “trampoline” needed for entering or exiting the kernel (syscalls, interrupts), while the full kernel mapping is only active in kernel mode. This blocks a large class of address-leakage side channels against KASLR, and later it directly maps to Meltdown mitigation logic since if kernel pages are simply not mapped during user execution, then transient reads from kernel addresses cannot retrieve real kernel data (because the addresses have no valid mapping at that point). The paper also discusses practicality concerns like minimizing TLB flushing cost via PCID and careful handling of global mappings, because without those details the overhead would be unacceptable on syscall-heavy workloads.

It is a clean OS-native mitigation, there is no need to rewrite applications, no need to reason about individual gadgets, the kernel just changes its mapping policy and the attack surface is immediately reduced. Also it became used broadly very fast, since the idea was essentially what production kernels shipped as PTI/KPTI once Meltdown happened [23].

The main limitation is the performance trade-off because switching page tables and paying the extra TLB pressure is expensive where OSes are sensitive, i.e. frequent syscalls and interrupts, plus, conceptually it only removes one big class of Meltdown-style leakage, but it does not eliminate the broader transient execution problem. [23].

3.7.2. WARD

WARD [6] starts from an observation they formalize as the Unmapped Speculation Contract (USC): if a physical page is not mapped in the current page tables, then it should not be possible to leak its contents through transient execution, because speculation cannot even name that memory via a valid virtual address. On top of that, WARD proposes a kernel design with two "worlds" per process: a Q domain (quasi-visible) where the kernel maps only memory that is safe to expose to that process and a K domain, where the full kernel mapping exists and where the usual heavy mitigations are enabled. Many syscalls can complete entirely in Q without paying the usual mitigation overhead at every entry or exit and only the syscalls that need sensitive data trigger a "world switch" to K (sometimes intentionally, sometimes transparently via a page fault). An important part of the paper is then about how to restructure kernel data so that Q can be genuinely "non-secret" (splitting structs into public/private parts, temporary mappings) and they evaluate the design in the sv6 research kernel with LEBench-style syscall workloads, showing that avoiding mitigations on the hot syscall path can recover an important percentage of performance.

The nice part is the conceptual framing, USC gives a clean argument for why mapping discipline can bound what is leakable and WARD uses that to design a kernel where the "no secrets are mapped" concept becomes an actual architectural invariant. It is also convincing because it does not just propose a piece, it shows a full kernel re-design path with concrete mechanisms (Q/K domains, world switch triggers, data-structure partitioning) and a performance evaluation that targets syscall-heavy overheads, where Meltdown mitigations hurt the most [6].

The approach is heavy on the kernel complexity, you pay with memory overhead, duplicated text mappings, extra page tables, and a lot of "make this structure public/private" engineering that does not scale for free to a full Linux-size codebase. Also, USC has boundaries, it mostly helps when the secret is in unmapped memory, and WARD explicitly ends up relying on Linux-style mitigations in the K domain. So it's not a universal TEA shield, it is a way to reduce the cost of the mitigations you still need [6].

3.7.3. Why 'pure software' has boundaries here

With Meltdown, software mitigations can feel like damage control because the vulnerability is tied directly to how the CPU performs permission checks relative to data fetch and speculative execution. While software mitigation we discussed in Sections 3.7.1 and 3.7.2 are able to narrow exposure and reduce practical exploitability, it becomes clear that they cannot single-handedly fix the underlying timing channels and speculation mechanisms, or at least not in a clean, universal way [6][30].

3.8. Positioning the thesis within the state of the art

The goal of the framework proposed in this thesis is to provide a practical, software-only hardening workflow that can be applied automatically to code regions that are likely exploitable under transient execution, then tuned to balance security coverage and performance cost.

Concretely, the pipeline is hybrid as it combines pattern-based detection of transient windows (e.g. V1-style bounds-check gadgets, V4/SSB store-load patterns and Meltdown-like faulting loads) with selective rewriting that generates multiple hardened variants of the same vulnerable region and also includes a search-based optimizer (genetic search over different configurations and transform mixes) that chooses a security–overhead operating point from empirical data.

This places the work close to the program analysis (selective rewriting) family (oo7 / Serberus) because the mitigation is led by a detector and applied only where needed, but it also intentionally borrows from data-flow hardening (SLH-like masking/dependency patterns) and barrier-based hardening (fences as a safe fallback) as alternative rewrite variants rather than committing to a single technique.

At the same time, it differs from papers like Specfuscator or "fence-everywhere" baselines

because it does not enforce one single policy, instead it treats mitigations as a configurable space.

Compared to control-flow manipulation & diversification approaches like Swivel, this thesis does less CFG reshaping, but it still adopts the core idea of manipulation and diversification at a finer granularity, by producing multiple implementations for each detected transient window and enabling configurations where the protected code region is duplicated and diversified, not kept unique, in fact the goal is to obtain multiple semantically equivalent, but structurally different implementations per window. The duplication is not only a software diversification mechanism in the traditional sense, it also has a microarchitectural implication because when the transient window is duplicated into multiple clones located at different instruction addresses, the attacker's measurements become less repeatable because execution enters equivalent gadget logic from different code locations over time, changing the instruction footprint and its interaction with caches and predictors.

For this reason, the framework is not fully orthogonal to microarchitectural footprint confusion either, while it does not implement OS-level channel suppression mechanisms like FlushTime, it does incorporate code transformations whose explicit goal is to degrade the quality of the recovery signal.

In contrast, it is very different from Meltdown OS mapping discipline (KAISER/WARD) because those approaches primarily increase security by changing what is mapped and when, while this thesis focuses on the code-level gadget stage.

In short, the framework is best seen as a composable middle layer: it can complement system-level mitigations, but its main contribution is an end-to-end automated workflow for detecting, rewriting, diversifying, and optimizing TEA mitigations in a unified way while also allowing some footprint-confusion transformations as part of the same variant space.

4 | Proposed technique and implementation

4.1. Framework overview

This thesis presents a software-only mitigation framework that performs an assembly-to-assembly rewriting, given an input `.s` file (typically given by `gcc -S / clang -S`), the tool produces an output `.s` file that is semantically equivalent but structurally different and hardened in selected regions that match transient-execution gadget patterns. The rewritten assembly can then be assembled and linked normally, without requiring a modified runtime or hardware changes.

Core intuition

TEAs exploit short sequences of instructions that access a secret and encode it into a measurable microarchitectural effect within a transient window. Attackers benefit from repeatability because executing the same gadget many times leads to a stable footprint that can be exploited statistically. The framework wants to take advantage of this property by making the execution of vulnerable regions less deterministic and less stable between runs, while preserving correct program outputs.

Variant-based hardening

For each detected transient window, the framework generates N semantically equivalent variants of that window, where variant 0 is always the original window, the other variants are produced by applying one or more local transformations.

The original window is then replaced by:

- a **lightweight runtime selector** that chooses a variant index
- the **variant blocks** (one per index)

- a **continuation label** that rejoins the original control flow

This design supports both diversification (reducing the stability of microarchitectural footprints) and security–overhead trade-offs (some variants may be more conservative, others less).

Transformation families (high-level)

The implemented transformation catalog can be presented as four families:

- **Speculation-stopping**: inserting serialization points at specific boundaries.
- **Data sanitization/hardening**: ensuring that transient badly computed indices or pointers are forced into a safe domain.
- **Dependency injection**: adding conservative dependencies to reduce the possibility of unsafe reordering in scenarios of data-speculation.
- **Structural perturbation**: changes that introduce confusion and diversify instruction mixes or scheduling opportunities.

4.2. Integration in the build/execution flow

The proposed framework is designed to be integrated into a standard build flow without modifying the compiler, assembler, linker or runtime (as shown in Figure 4.1). The tool is an intermediate pass on assembly emitted by the compiler because it takes an input .s file produced from a translation unit (in the cases considered in Chapter 6 the original files were always `.c` so that is what we will use going forward) and rewrites only the regions that match transient-execution window patterns, emitting a mitigated `.s` that then can be assembled and linked normally.

In terms of artifacts, the integration point can be summarized as follows:

1. **Compilation to assembly**: a conventional compiler (e.g. GCC/Clang) translates `.c` into `.s`.
2. **Rewriting stage** (this framework): the emitted `.s` becomes the input of the framework which performs detection and window rewriting, outputting a new `.s` according to the chosen parameters or policy.
3. **Assembling and linking**: the rewritten `.s` is passed to the system assembler and linker to produce the final executable.

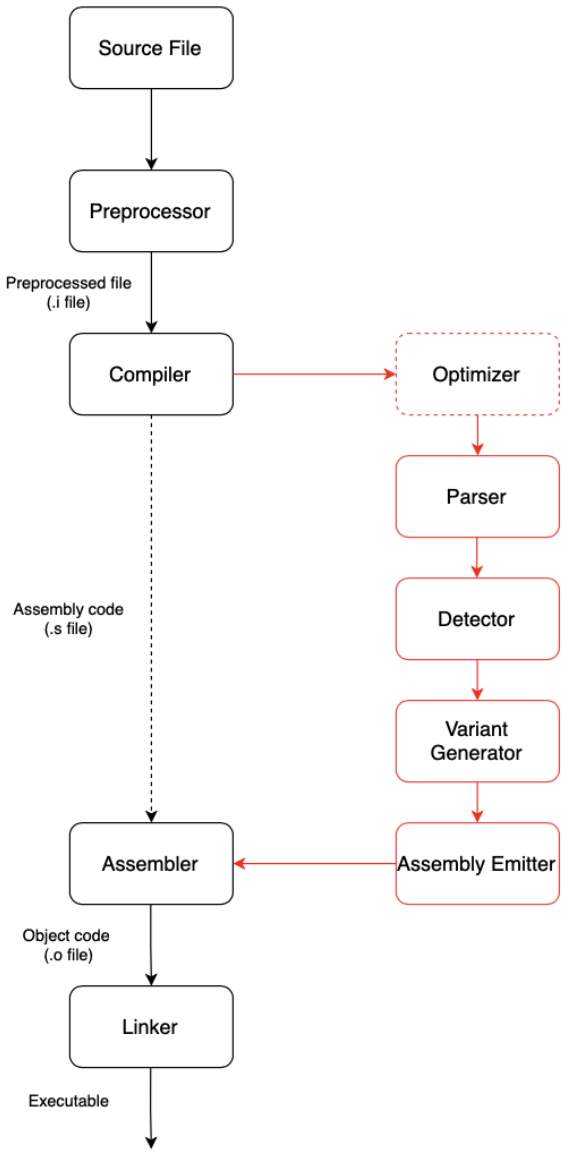

Figure 4.1: Position of the framework within a conventional compilation pipeline.

Syntax and toolchain assumptions

The current prototype targets x86-64 assembly in AT&T syntax, matching the typical output of `gcc -S` / `clang -S`. The parser is intentionally lightweight and in fact, it assumes compiler-like formatting (labels, directives, and instruction lines), this design choice was made to keep the rewriting stage simple for the intended setting (compiler generated gadgets and windows) at the cost of portability to different ISAs or assembler syntaxes.

Dependencies

A key advantage of placing the framework at the `.s` level is that the final binary does not require a dedicated runtime library since variant selection is implemented directly inside the rewritten assembly. This keeps deployment simple because the output remains a self-contained assembly file that can be assembled and linked with the standard system toolchain.

4.3. Internal pipeline

The implementation is structured as a modular pipeline, at a high level, the rewriting flow for an input file is the following:

1. **Parsing and normalization**: the `.s` file is parsed into an internal representation (grouped by function) that keeps the original instruction order and important assembler directives, while also exposing mnemonics and operands in a form useful for analysis and transformation. Each instruction gets also extended with newly added metadata used later in the pipeline (e.g. transient-window flags and structural tags).
2. **Transient window detection**: for each function, a configurable set of detectors performs a linear scan and matches patterns relevant for TEAs, detected windows are identified with their boundaries and the corresponding instructions are annotated so that in later stages the pipeline can operate on specific window intervals and avoids a re-running of the analysis.
3. **Variant generation and code injection**: for every selected (non-overlapping) window the framework generates a (configurable) number of variants, the first variant is an exact copy of the original window while the remaining variants are obtained by applying one or more local transformations sampled from a weighted distribution. The original window gets replaced with a runtime selector block, the variant

blocks and a shared continuation label (as previously stated in Section 4.1).

4. **Writing of the mitigated assembly file**: the final internal representation is serialized back into the output `.s` file maintaining the original structure and directive placement while including the newly written selector and variant blocks along with any required helper code so that the output can be assembled and linked correctly.

The following subsections detail these steps in order, Section 4.3.1 introduces the parsing layer and the internal model, Section 4.3.2 describes the detector subsystem and the rules associated to window formation and Section 4.3.3 tells how variants are produced and injected while preserving architectural correctness and Section 4.3.4 discusses the writing process which results in the protected `.s` output.

Concrete examples are provided in the Appendix to complete the description of the pipeline, specifically, Appendix A includes examples of how input assembly is parsed and mapped into the internal objects and how a detected window is rewritten to include the selector and variant blocks.

4.3.1. Parsing and IR model

Because the framework operates on compiler-emitted assembly, the first step is to convert the `.s` file into a lightweight internal model that is structured enough in order to support detection and rewriting, but conservative enough to preserve all lines that are toolchain-sensitive (directives, labels, CFI annotations) in their original order.

Internally, the parsed assembly is stored through two conceptual building blocks:

1. **Function container** (Function-like class): this object represents one top-level symbol scope (e.g. main:), this structure is needed since detectors and rewriters operate per function, the container keeps:
   - a function identifier (the entry label name)
   - a list of header directives that appear immediately before the entry label (e.g. visibility or type directives written by the compiler)
   - an ordered list of line records representing the body
2. **Line record** (Instruction-like class): every line belonging to a function body is represented as a record.
   For actual machine instructions, the record stores:

- the original raw line (to preserve the source text)
- a parsed mnemonic
- a list of operand tokens (kept in AT&T textual form)
- an optional trailing comment

For non-instruction lines (labels, directives inside the function or lines the parser does not interpret like comments) the record still exists but carries only the raw line, so it can be emitted verbatim.
Later stages enrich these records with analysis metadata (e.g. "it belongs to a transient window" and a tag marking if on that instruction a structural transformation was already applied).

This model is intentionally not a compiler IR since it does not build a CFG, does not compute dataflow, and does not want to interpret full instruction semantics, it provides just enough structure to support pattern matching and local rewriting.

How parsing works

The .s file is read line-by-line and each non-empty line is classified into one of the following categories:

1. **Function entry label**: a "top-level" label of the form `name:` (with name not starting with .) starts a new function container, any pending header directives accumulated immediately before the label are attached to that function.
2. **Assembler directive**: lines starting with `.` are treated as directives, if they appear outside a function scope, they are buffered as pending header directives while if they appear inside a function, they are saved as body line records to preserve exact positioning (this is essential for .cfi_* sequences).
3. **Instruction line**: lines that match the typical "mnemonic + operands + optional comment" pattern, they are parsed into a mnemonic and an operand list but the raw line is still saved as it is.
4. **Fallback/raw line**: anything else (including internal labels or unusual formatting) is stored as a raw line record inside the current function.

A conservative end-of-function marker (e.g. the compiler's CFI end directive) is used to close the current function.

Operand tokenization policy

Detectors and transformations must not break AT&T memory operands such as `(%rbx,%rcx,8)` by splitting on commas inside parentheses. In fact operand splitting follows a simple but effective rule: *split on commas only when not inside parentheses.* This yields stable tokens for registers, immediates and memory addressing forms without needing a full operand grammar.

4.3.2. Transient window detection

After parsing, the framework identifies candidate transient execution windows through a set of independent pattern-based detectors, the goal of this stage is to conservatively locate instruction subsequences that resemble TEA gadgets, like sequences containing a plausible speculation trigger and at least one memory access that could participate in the encoding.

Detection workflow

Detection is performed per function on the instruction stream produced by the parser, detectors share a common interface, each one examines the instruction list and when it recognizes its pattern at a given position, it returns a window described by:

- a **start index** and **end index** in the instruction list
- a **pattern identifier** (used later for reporting and analysis)

When a window is found all the instructions in the corresponding interval are flagged with a boolean flag ("belongs to a transient window"), in addition the detector stage forces structural safety boundaries because windows are kept intra-basic-block by stopping at "real" labels and at control-flow boundaries so that later rewriting does not need to duplicate labels or cross instructions like returns and jumps.

A single global parameter, the window size, tells how far a detector can look forward (or backward, for indirect branches), this bounds runtime and also false positives since we are restricting pattern matching to short sequences.

Detector 1 (Spectre V1-style)

This detector identifies the classic Spectre V1 structure where a bounds or condition check is followed by a conditional branch but speculation may transiently execute the fall-through path and perform a sensitive load or an encoding access. The pattern the

detector looks for is as follows:

1. An instruction of type `cmp` or `test` acts as the check.
2. Within a bounded lookahead, a conditional jump (`jcc`) is found.
3. Within another bounded lookahead after the `jcc`, at least one instruction with a memory operand is found (the detector extends the window up to the last memory access like this within the lookahead).

The detector also showcases a specific behaviour in terms of window boundaries and constraints:

- The scan stops at basic-block boundaries and does not cross "real" labels.
- The scan stops at hard control-flow boundaries like unconditional jumps and returns to avoid saving a window found between discontinuous code.
- The detected window starts at the `cmp` or `test` and ends at the last qualifying memory access found after the branch.

Detector 2 (Spectre V4-style)

Spectre V4-style attacks rely on the processor transiently executing a load before an older store has been correctly forwarded, potentially observing a stale value in a transient manner. The pattern the detector looks for is as follows:

1. The start is a store recognized in AT&T form as a move-like instruction whose destination operand is memory.
2. Within a certain lookahead (bounded by the window size parameter), the detector searches for the first subsequent load identified as a move-like instruction whose source operand is memory.
3. A simple heuristic is applied: the store and load are considered potentially related when their memory operands share the same base register in the addressing mode.

For this specific detector the following principles are also applied to avoid many possible false positives:

- Stack accesses are excluded (memory operands based on `%rbp` or `%rsp`), since instructions containing them are plenty.
- RIP-relative accesses are excluded to avoid flagging constant or global addressing patterns which are common in position-independent code but basically never match

the gadget structure we are looking for.

The detector also showcases a specific behaviour in terms of window boundaries and constraints:

- The scan does not cross basic-block boundaries (labels) and does not cross control-flow boundaries (including conditional branches and calls), this is done to keep the window local and to avoid ambiguous reordering across divergent control paths.
- To reduce overlap and explosion of the code size, the detector does not start a window at an instruction that has already been annotated as part of another transient window (greedy non-overlap).

Detector 3 (Meltdown-like)

Meltdown-style gadgets have two key components: a potentially faulting load that transiently brings secret data into a register, and a subsequent secret-dependent access (usually an array access) that encodes the secret into cache state. Static analysis cannot determine if a load will fault at runtime, so the detector's goal is to identify candidates with the right data-dependence structure.

1. The start is a move-like load from non stack and non RIP relative memory into a general purpose register.
2. In the bounded forward window the destination register of the anchor load is marked tainted and taint is propagated to subsequent instructions when their destination depends on the tainted sources.
3. The detector looks for a later instruction that performs a memory access whose address uses a tainted register, this indicates a data-dependent encoding step.

Here are some rules for taint propagation used in this detector:

- If an instruction computes a destination register using any tainted source register, the destination becomes tainted.
- If an instruction overwrites a destination register without using tainted sources (e.g. a simple move from untainted inputs), the taint for that destination is removed.
- Common zeroing instructions (e.g. `xor reg, reg`) are basically treated as explicit taint clears.

The detector also showcases a specific behaviour in terms of window boundaries and constraints:

- The scan stops at basic-block boundaries (labels).
- It stops at returns, unconditional jumps and calls, however, it allows conditional branches inside the window since real PoCs often contain them but still keep the load-encode dependence in the same local region.
- Like the Spectre V4 detector, it applies greedy non-overlap.

Once a load to dependent-memory structure is found, the detector can optionally consider additional signals to make sure the window is correct:

- page-stride shaping operations (e.g. shifting by 12 or multiplying by 4096, these are consistent with cache-line or page indexing strategies)
- cache or timing primitives (e.g. clflush)
- TSX related markers (e.g. transaction begin/end/abort)

These hints do not define the window, they are used only to reduce ambiguity when multiple possible windows are found close to each other.

Output of the detection stage

At the end of this stage, each function carries a list of detected windows (with start, end and pattern) and some annotations for each instruction which state if the instruction belongs to a window. This stream is the input of the rewriting phase.

4.3.3. Variant generation strategy

Once transient windows have been identified, the framework performs localized diversification by replacing each detected window with: a selector block, a set of N alternative implementations ("variants") of the same window and a single continuation label that restores the original control-flow by jumping to the first instruction after the window. This design keeps the scope of the transformation strictly intra-function and window-local, which is important to maintain correctness and the ability to perform debugging.

Variant model

For each function containing at least one detected window, the variant generator first normalizes the detector output into one list of windows for each function and then applies two pre-processing steps before any rewriting.

First, it enforces a conservative non-overlap policy: windows are sorted by start index and any window that overlaps (even by a single instruction) with a previously selected one is discarded, this prevents nested or overlapping rewrites and provides stable semantics for window boundaries.

Second, the selected windows are processed in reverse order (from the bottom of the function toward the top). The reason is since rewriting replaces a window with a larger block of code, operating bottom-up avoids having to recompute indices and offsets for windows that appear earlier in the instruction list.

After selecting a window, the generator creates a list of variants:

- Variant 0 is always an exact copy of the original window instructions.
- Variants 1...N−1 are produced by applying a sequence of `transforms_per_variant` transformations to the original window, the transformation for each step is sampled from a configurable probability distribution (which we call "weights").
- A special configuration flag allows forcing all variants to be identical copies of the original window (`same_variants`). This is useful because it isolates the overhead of the selector mechanism from the overhead introduced by transformations and as we will see in Section 4.4.1, it can be seen as a transformation of its own.

In terms of configurability, the generator supports number of variants, number of transformations chained per variant, an optional random seed for reproducibility and an optional weight map that biases the selection of transformations. If weights are not provided (or are degenerate), the generator falls back to a uniform distribution over the enabled transformation set.

Finally, not all transformations have the same resource needs, some rewrites require one or more temporaries that are considered safe to use within that function (e.g. for address arithmetic or branchless masking). For these, the generator computes a set of safe temporary registers once per function and passes them only to transformations that explicitly require them.

Runtime selector block

To decide which variant executes at runtime, the generator writes a compact selector block immediately at the original window position. The selector has two goals: be self-contained (no global state required) and preserve the architectural state that the surrounding code

expects. The implemented selector follows a simple pattern:

1. It saves the registers it will clobber (notably `%rax`, `%rcx`, `%rdx`).
2. It reads a timestamp using `rdtsc`, mixes the two halves (`xorq %rdx, %rax`), and computes an index in 0,num_variants−1 using integer division (remainder in `%edx`, then moved into `%eax`).
3. It restores `%rcx` and `%rdx` before branching to the selected variant label, leaving `%rax` saved on the stack for later restoration by the variant prologue.
4. It performs a compare/jump chain (`cmpl` + `je`) to transfer control to the chosen variant block (with a fallback unconditional jump).

Each variant block begins with: a unique label, a `popq %rax` to restore the saved `%rax`, the variant body instructions and a `jmp` to a common continuation label (one per rewritten window), this structure ensures that control always restarts at a single point, so that the remainder of the original function executes without changes and does not need to be duplicated.

Because the selector and variant blocks introduce extra pushes/pops and control-flow edges, the generator includes two practical protection techniques.

The first one is a red-zone / leaf-function stack safety: in x86-64, leaf functions may use the red-zone (space below `%rsp`) for locals without moving the stack pointer, the selector's pushes can overwrite such locals. The generator therefore detects this case and inserts a small stack padding adjustment (`subq $PAD, %rsp`) before the selector, then compensates it at the start of each variant (immediately after restoring `%rax`). This way, even if a path leaves the window early, the restoration is not skipped, the stack remains consistent and we do not have any problem.

The second one is avoiding to apply transformations on already rewritten instructions: several transformations tag the instructions they insert as structural so that subsequent transformation steps (working on the same variant) ignore those lines and cannot be applied on them, this reduces the risk of composing transformations in a way that would cause a change in the behavior of the final program.

4.3.4. Output writing

After window rewriting, the framework serializes the modified internal representation back into a valid `.s` file that can be used by a standard assembler and linker, this stage is made conservative intentionally since it aims to preserve the original structure and ordering of the compiler output while including the additional code needed for variant selection and the transformed window bodies.

Emission strategy

The writer passes through the internal model in source order and emits an assembly file with the same structure as the input:

1. **Preamble and function headers**, any directives that belong to the global preamble or were associated with a function header are written before the corresponding function label, keeping their original order.
2. **Function body**, each function is written as an ordered list of line records, the writer distinguishes between:
   (a) **raw lines** (labels, directives, and any non-parsed lines), which are emitted verbatim to avoid disrupting assembler metadata (notably .cfi_* directives)
   (b) **instruction records**, which are re-serialized in a canonical "mnemonic operand[, operand...]" form possibly followed by a saved comment
3. **Postamble and trailing directives**, directives that appear after the function body in compiler output are emitted after the function, maintaining their position.

This policy ensures that the output remains close to the assembly which was originally provided by the compiler, limiting the risk of toolchain incompatibilities that could happen from situations like global reformatting or directive relocation.

Labeling and structural consistency

Window rewriting includes new basic blocks (selector, variants, continuation), to keep the emitted file correctly formed and avoid compilation errors all newly generated labels are constructed to be unique within the translation unit, furthermore we also know from Section 4.3.3 that each variant block ends with an explicit branch to a single continuation label, so we are sure that control-flow always restarts at a well-defined point and the remainder of the function can be written unchanged.

If a transformation requires out-of-line helper sequences (e.g. pointer sandboxing, which we will describe in Section 4.4.3), the writer writes these blocks in a dedicated area (typically at the end of the file), ensuring they are emitted exactly once and are reachable via the newly written branches.

Overall, the emission stage completes the framework, the mitigated output is a standard `.s` file that can be assembled and linked through an unmodified toolchain, while embedding the selector-based variant mechanism and the transformed window bodies introduced by the previous stages.

4.4. Transformation catalog

This section describes the transformation catalog implemented by the framework and explains how each transformation modifies the instruction stream inside a detected transient window. Unlike the previous sections which focused on the end-to-end pipeline, the goal here is to provide a transformation-level view, basically what local assumptions each rewrite makes, which instruction patterns it expects and how it changes the transient execution opportunities available to an attacker.

All transformations are designed to be local and composable, since they operate on a bounded window interval, keep the architectural semantics of the original code region and may write a small amount of additional structure (extra instructions, fences, or helper logic) to modify the microarchitectural behavior that TEAs rely on. In the prototype, transformations are applied only in the boundaries produced by the detectors. When a transformation cannot be applied safely, it conservatively becomes a no-op so that the resulting variant remains correct.

To make the effect of each transformation concrete, the remainder of this section is organized around before/after assembly code snippets. First, for each of the targeted TEA classes, a representative transient window identified by the corresponding detector is shown, then, for every transformation, the same window is presented again with the transformation applied in isolation only once, so that the reader can directly observe how the rewrite changes the transient window in practice and which part of the gadget structure is being disturbed.

Spectre V1 candidate window

Listing 4.1: Spectre V1 detected window

```
1 cmpq %rcx, %rax
2 jae LBB0_2
3 movq -8(%rbp), %rcx
4 leaq _array1(%rip), %rax
5 movzbl (%rax,%rcx), %eax
```

Spectre V4 candidate window

Listing 4.2: Spectre V4 detected window

```
1 movq %rdi, -8(%rbp)
2 movq _memory_slot_ptr@GOTPCREL(%rip), %rax
3 movq (%rax), %rax
```

Meltdown-US candidate window

Listing 4.3: Meltdown detected window

```
1 movb (%rdx), %al
2 shlq $0xc, %rax
3 jz 92b
4 movq (%rcx, %rax, 1), %rbx
```

4.4.1. Window duplication/variant injection

Applies to: any detected transient window.
Goal: reduce gadget repeatability by making sure that the same vulnerable region does not always execute an identical instruction stream across runs.
Rationale (why it is a transformation): even without applying any additional protection inside the window, duplicating the window itself changes the execution behavior from the attacker's perspective, TEA exploitation typically relies on repeating the same gadget many times to accumulate a stable microarchitectural signal. Window duplication breaks this assumption, so repeated triggers no longer deterministically execute the exact same basic block at the same code location. In this sense, duplication is a transformation on the transient window because it modifies the window's microarchitectural footprint, while preserving architectural semantics.

Rewrite (high-level): the original window is replicated into N semantically identical copies, each copy executes the same instruction sequence and then restarts the original control flow at the same continuation point, at runtime one of the copies is chosen for execution.

Listing 4.4: Spectre V1 baseline window after duplicating the window.

```
## Variant 0
.Lvictim_function_win0_var0:
    popq %rax
    addq $32, %rsp
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
    jmp .Lvictim_function_win0_continue
## Variant 1
.Lvictim_function_win0_var1:
    popq %rax
    addq $32, %rsp
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
    jmp .Lvictim_function_win0_continue
...
```

Expected effect on the transient window: the transient behavior remains functionally equivalent but the microarchitectural observations become less consistent because the window's execution is spread across multiple code locations and paths, as a result of this we obtain increased noise and possibly raise the number of samples required to recover the secret reliably.

4.4.2. Speculation-stopping

Speculation-stopping transformations aim to terminate or bind speculative execution at a boundary that is known to open a transient window, conceptually, these transformations trade performance for robustness by adding serialization points.

Fencing after conditional branch (`transform_fence_after_jcc`)

Applies to: Spectre V1-style windows (the window containing cmp/test → jcc → memory access).

Goal: prevent execution from moving past the conditional branch until the branch condition is resolved, removing the key "speculate past bounds check" capability.

Preconditions/applicability: the detected window must contain a conditional branch (jcc) that acts as the guard for subsequent memory accesses.

Rewrite (high-level): insert an `lfence` immediately after the guarding jcc on the fall-through path (i.e., inside the transient window region, right after the conditional jump instruction).

Listing 4.5: Spectre V1 baseline window after inserting lfence.

```
1 cmpq %rcx, %rax
2 jae LBB0_2
3 lfence
4 movq -8(%rbp), %rcx
5 leaq _array1(%rip), %rax
6 movzbl (%rax,%rcx), %eax
```

Expected effect on the transient window: speculative execution past the check is suppressed (or significantly reduced depending on microarchitecture), so secret-dependent loads and their encoding accesses are much harder to execute transiently.

Fencing between store and load (`transform_fence_between_store_load`)

Applies to: Spectre V4 / Speculative Store Bypass windows (store → load sequence within the same local region).

Goal: prevent the load from being executed (transiently) before the store becomes globally visible, reducing the chance that the load observes a stale value.

Preconditions/applicability: the window must contain a store followed by a later load that is a possible SSB pattern (as identified by the detector).

Rewrite (high-level): insert an `mfence` between the store and the following load.

Listing 4.6: Spectre V4 baseline window after inserting mfence.

```
1 movq %rdi, -8(%rbp)
2 movq _memory_slot_ptr@GOTPCREL(%rip), %rax
3 mfence
4 movq (%rax), %rax
```

Expected effect on the transient window: the store-load reordering window is effectively closed, reducing the opportunity for speculative bypass of store forwarding or disambiguation.

Fencing to cut a load-encode dependency chain (`transform_fence_cut`)

Applies to: Meltdown-like windows (faulting load candidate → data-dependent memory access).
Goal: stop speculative propagation of the loaded value into later address computations used for encoding.
Preconditions/applicability: the window must contain a load that is used in a later memory access whose address depends on that value.
Rewrite (high-level): insert an `lfence` after the candidate load (or just before the dependent encoding access, depending on the instruction pattern) so that the architectural resolution of the fault/permission outcome becomes a hard boundary for further execution.

Listing 4.7: Meltdown baseline window after inserting lfence.

```
1 movb (%rdx), %al
2 lfence
3 shlq $0xc, %rax
4 jz 92b
5 movq (%rcx, %rax, 1), %rbx
```

Expected effect on the transient window: the critical "load secret → use secret as address" sequence gets broken, preventing secret-dependent encoding from executing transiently.

4.4.3. Data sanitization/hardening

Data sanitization/hardening transformations do not stop speculation, they try to ensure that even if speculative execution proceeds, the transient computation is forced into a safe domain (e.g. safe indices, safe pointers), so the attacker cannot drive the encoding step toward observable microarchitectural state.

Lightweight index masking (`transform_index_masking_light`)

Applies to: Spectre V1-style bounds-check windows (array index used after a conditional guard).

Goal: ensure that if the bounds check is bypassed transiently, the index used for the subsequent memory access is forced to a safe value (typically zero or within bounds) removing attacker control on the access pattern.
Preconditions/applicability: the window must contain a check + conditional branch and a later memory access that uses an index-like register/value in address computation.
Rewrite (high-level): compute a mask derived from the outcome of the guard and apply it to the index (or the pointer derived from it) before the memory access, the rewrite is designed to be branchless in order to avoid introducing new control-flow.

Listing 4.8: Spectre V1 baseline window after lightweight index masking.

```
1 cmpq %rcx, %rax
2 jae LBB0_2
3 xorq %r11, %r11
4 cmovb %rcx, %r11
5 movq -8(%rbp), %rcx
6 leaq _array1(%rip), %rax
7 movzbl (%rax,%r11), %eax
```

Expected effect on the transient window: speculative execution may still happen, but the accessed address is not controlled by the attacker, this prevents secret-dependent cache encoding through an out-of-bounds index.

Pointer domain restriction (`transform_pointer_sandboxing`)

Applies to: Meltdown-like windows.
Goal: ensure that transient dereferences cannot reach sensitive address ranges by rewriting the pointer, so that unsafe high address bits are redirected into a dedicated sandbox region.
Preconditions/applicability: the window must contain a pointer-carrying register used as the base of a memory operand (or used to compute this base).
Rewrite (high-level): insert a short sequence that transforms the pointer into a sandboxed pointer (e.g. by constraining the address range and/or selecting between the original base and a known mapped sandbox page), the transformation requires auxiliary data (the sandbox page) to be written once in the final assembly.

Listing 4.9: Meltdown-like baseline window after pointer sandboxing.

```
1 leaq __gg_meltdown_sandbox(%rip), %r10
2 testq %rdx, %rdx
3 cmovs %r10, %rdx
```

```
movb (%rdx), %al
shlq $0xc, %rax
jz 92b
movq (%rcx, %rax, 1), %rbx
...
...
...
# Meltdown sandbox page generated at the end of the file
__gg_meltdown_sandbox:
    .skip 4096
    .text
```

Expected effect on the transient window: even if a fault/permission check happens transiently, the access is redirected into attacker-irrelevant memory preventing the secret extraction.

Note: given that this transformation applies only to Meltdown-like windows and emits specific auxiliary directives at the end of the rewritten assembly file that are compatible with the GNU assembler (GAS), it should only be enabled when the input `.s` file is assembled with a GAS-compatible toolchain (e.g. the standard Linux x86-64 build pipeline). Otherwise the transformation may still be applied, but the final file may fail to assemble due to unsupported directives.

4.4.4. Dependency injection

Dependency-injection transformations try to reduce unsafe transient behaviors by introducing conservative data dependencies that limit the core in terms of how it can execute a later operation before an earlier one gets resolved, compared to fences, these are often lower overhead but their effectiveness is more microarchitecture-dependent.

SSB dependency barrier (`transform_ssb_dependency_chain_barrier`)

Applies to: Spectre V4 / SSB-style store→load windows.

Goal: reduce the possibility that the load can execute using an incorrect (stale) value by forcing it to depend on a value that cannot be obtained until the relevant store computation is known.

Preconditions/applicability: the window must contain a store followed by a later load that is a probable SSB pattern, the rewrite must have access to a safe temporary register when needed.

Rewrite (high-level): insert a small (semantics-preserving) dependency chain such that the later load's effective address becomes data-dependent on a value tied to the store's address or base computation.

Listing 4.10: Spectre V4 baseline window after injecting a dependency barrier.

```
movq %rcx, (%rax)
movq %rax, %r10
movq _memory_slot@GOTPCREL(%rip), %rax
pushfq
andq $0, %r10
popfq
movq (%rax,%r10), %rax
```

Expected effect on the transient window: the store→load window is reduced because the load cannot be issued as early without the dependency being satisfied.

4.4.5. Structural perturbation

Structural perturbation transformations are mostly used to diversify the low-level structure of the window, they do not necessarily force a strict speculation boundary, instead, they modify timing, dependency shape and instruction mix, which can reduce the stability of the microarchitectural footprint that attackers tries to exploit.

For these transformations, we can consistently demonstrate the effect on the Spectre V1 baseline window (for readability), since the perturbations are TEA-agnostic and the goal is to show the mechanics of the rewrite rather than a TEA-specific predicate.

NOP padding (`transform_nop`)

Applies to: any detected window (illustrated on the Spectre V1 baseline window).
Goal: perturb timing and slightly change scheduler behavior without changing the architectural state.
Preconditions/applicability: none beyond having an insertion point in the window that is safe.
Rewrite (high-level): insert one or more NOP instructions inside the window leaving the rest of the original instruction sequence unchanged.

Listing 4.11: Spectre V1 baseline window after NOP padding (applied twice).

```
cmpq %rcx, %rax
```

```
jae LBB0_2
nop
movq -8(%rbp), %rcx
leaq _array1(%rip), %rax
nop
movzbl (%rax,%rcx), %eax
```

Expected effect on the transient window: the transient window remains logically similar but its timing and microarchitectural footprint is disturbed, which can increase noise in repeated measurements.

Address computation splitting (`transform_lea_split`)

Applies to: windows containing non-trivial addressing computations (illustrated on the Spectre V1 baseline window where a memory operand can be rewritten).
Goal: change the μop composition and dependency structure of address generation by rewriting a complex address expression into multiple simpler steps.
Preconditions/applicability: the target memory operand must be expressible as a split sequence using a temporary register.
Rewrite (high-level): replace a single complex addressing form with a sequence that computes the effective address in one or more intermediate steps (using `lea` into a temporary), then perform the load/store through the computed address.

Listing 4.12: Spectre V1 baseline window after LEA splitting.

```
cmpq %rcx, %rax
jae LBB0_2
movq -8(%rbp), %rcx
leaq _array1(%rip), %rax
leaq (%rax,%rcx), %r10
movzbl (%r10), %eax
```

Expected effect on the transient window: the memory access still occurs architecturally but the address-generation and dependency chain are different, altering the microarchitectural side effects.

Reordering independent moves (`transform_reorder_movs`)

Applies to: windows containing multiple independent mov-like instructions (illustrated on the Spectre V1 baseline window).

Goal: diversify scheduling opportunities by reordering operations that do not have data dependencies between them.
Preconditions/applicability: there must be at least two move-like instructions in the window that are independent under a conservative dependency check (no shared registers).
Rewrite (high-level): swap the order of independent mov instructions while maintaining semantics.
Expected effect on the transient window: architectural behavior is not modified but execution timing and transient overlap patterns could be different, this can reduce the stability of attacker observations during repeated runs.

5 | Genetic optimization

Chapter 4 introduced the mitigation framework and showed that its behavior depends on several parameters, yielding a large configuration space with non-trivial security–overhead trade-offs. This chapter starts by outlining this configuration surface, and then presents a genetic optimization approach to automatically search the space and select configurations that best satisfy a target trade-off, especially when protecting multiple TEA variants under explicit overhead or protection constraints.

5.1. Configuration surface

This section summarizes the configuration interface exposed by the prototype and clarifies why it induces a large search space. Rather than enforcing a single fixed mitigation policy, the framework is parameterized: the chosen settings influence which transient windows are detected and rewritten, how many alternative implementations are produced per window, and which transformations are used to construct non-baseline variants. Even when each parameter ranges over a small discrete domain, their combinations quickly yield a large configuration space, motivating the need for automated search.

5.1.1. Parameters defining the search space

The framework exposes a set of parameters that define the configuration space, they can be classified into:

- **detection parameters** (window size and enabled detectors)
- **diversification parameters** (number of variants and transformations composed per variant)
- **transformation selection parameters** (subset of enabled transformations and their weights)
- **reproducibility and control parameters** (seed and "same variants" mode)

5.1.2. Cost drivers induced by the configuration

The parameters above do not only change the shape of the rewritten code, they also influence the overhead sources:

- Runtime overhead is driven by the number of rewritten windows (how often a selector executes), the presence of serialization-heavy transformations (e.g. fences) and the amount of extra work introduced inside variants (additional instructions, dependency chains, sandboxing logic).
- Code size and cache pressure increase with the number of variants per window and with helper code/data that some transformations require, which can indirectly affect performance beyond the transformed windows themselves.
- Generation time grows with the number of windows and with the amount of variants generated for each window.

These drivers make the choice of configuration a multi-objective optimization problem, which motivates the genetic search strategy introduced in the next sections.

5.2. Genetic algorithms (brief explanation)

A *genetic algorithm* (GA) is a population-based, stochastic optimization method inspired by biological evolution [24]. Instead of refining a single candidate, a GA maintains a population of candidates and applies repeatedly selection and random variation to discover better solutions according to a fitness function, this is particularly useful when the search space is large like in our case [21].

A canonical GA iteration proceeds in "generations", first, each individual (candidate solution) is encoded as a chromosome (a vector of genes), the algorithm evaluates each chromosome with a fitness function, then uses a selection strategy to drive reproduction toward individuals with higher fitness. From selected parents, offspring are generated through variation operators (crossover and mutation) and a replacement strategy forms the next population (often with elitism, meaning a small number of best individuals are copied unchanged). The algorithm stops when it reaches a preset budget or when improvements stop [43].

The chromosome representation (the genotype-to-phenotype mapping) is an important design choice because it determines what it means to "mix" or "perturb" a solution. Genes can be binary values, integers, real numbers or more structured objects, a representation

should make the evaluation well-defined and should let variation operators be able to produce meaningful neighbors and ideally, valid solutions with high probability. The encoding shapes the search and strongly influences convergence behavior and the degree to which small genetic changes correspond to small semantic changes in the candidate solution [43].

Selection defines the "exploitation" pressure of the GA, common choices include fitness-proportionate selection (sensitive to scaling), rank-based selection (more robust to scaling) and tournament selection [7] (simple and widely used, with selection pressure controlled by tournament size), variation instead provides “exploration”. Crossover recombines genetic material from two parents and is often used as a way to preserve and recombine useful partial structures. Mutation introduces random changes to keep population diversity and reduce premature convergence to suboptimal regions, basically it tends to explore new combinations of already discovered structures, while mutation ensures continued innovation along with local exploration [43].

In conclusion, many real applications require handling constraints and multiple objectives, constraints can be addressed by rejecting invalid individuals, repairing them into feasible ones, designing encodings or operators that preserve feasibility or adding penalty terms to the fitness so that violations are penalized while still permitting exploration near feasibility boundaries [11]. When objectives conflict (e.g. maximize quality while minimizing cost), it is common to treat the problem as multi-objective optimization and aim to approximate a Pareto frontier rather than reducing everything to a single scalar score [14].

5.3. GA design

The implementation follows a standard evolutionary loop built on top of PyGAD [17]: a population of candidate configurations is evolved over multiple generations, where each candidate is evaluated through the surrogate-based fitness pipeline and after that it is improved using selection and variation. The design choices in this part of the system are aimed at handling a mixed search space which includes both discrete and continuous parameters, keeping individuals valid at every generation and maintaining enough exploration to avoid early convergence to configurations that are only locally optimal.

The GA is configured with a tournament-based [7] parent selection strategy, single-point crossover and random mutation applied to a fraction of genes. A small number of parents are preserved across generations (elitism via parent retention) to reduce the risk of losing

the best configuration due to stochastic variation. Reproducibility is supported by using an explicit random seed, so that the same experimental setup can be re-run when needed. The next subsections detail how candidate solutions are represented, how variation operators are applied to that representation, how crossover is performed while preserving validity and how constraints are integrated into the optimization process.

5.3.1. Chromosome representation

Each GA individual encodes a candidate configuration as a fixed-length chromosome that can be decoded into the configuration object used by the rest of the pipeline, the chromosome is organized as a compact vector with three "structural" genes followed by a set of continuous genes representing transformation preferences:

- `num_variants`: chooses how many variants are generated per transient window.
- `transforms_per_variant`: chooses how many transformations are composed to build each non-original variant.
- `same_variants`: a binary flag (0/1) that enables generating copies instead of diversified variants.
- `w_*` genes: one real-valued gene per transformation weight column, representing the relative preference for selecting that transformation during variant construction.

The representation is attack-agnostic, it expresses a single global policy, while leaving any attack-specific interpretation of weights to later stages of the evaluation pipeline, the number of weight genes is not hardcoded, it is derived from the available `w_*` columns in the experimental dataset, so the chromosome length scales automatically with the set of modeled transformations.

To make this representation compatible with generic GA operators, the first two genes are represented in the GA search space as bounded real values and then converted back to integers during decoding to the allowed ranges. The `same_variants` gene is constrained to the discrete set 0, 1. Weight genes are bounded non-negative real values and are decoded as raw weights without requiring them to sum to one..

5.3.2. Mutation operators

The implementation relies on PyGAD's built-in "random" mutation [17], applied at the gene level, at each generation, mutation is triggered by selecting a fixed percentage of

genes in the offspring (configured via `mutation_percent_genes`) and resampling each selected gene from its admissible domain (`gene_space`). This choice keeps mutation simple and compatible with the mixed nature of the chromosome, furthermore, the same generic mutation mechanism produces different effects depending on the gene type:

- Structural parameters (`num_variants`, `transforms_per_variant`), these genes are mutated by drawing a new value within their bounded interval, since PyGAD treats them as continuous values, it applies rounding and clamping during decoding, ensuring that mutated candidates always map to valid integer settings. In practice, these mutations provide "coarse" exploration by changing the breadth (number of variants) and depth (number of transformations per variant) of diversification.
- Boolean switch (`same_variants`), this gene is constrained to the discrete set 0, 1, so mutation effectively switches between "generate diversified variants" and "allow duplicates", this gives the optimizer a low-cost way to explore different policies.
- Transformation preference weights (`w_*`), each weight gene is mutated by resampling within a fixed non-negative interval (in the current implementation, [0, 1]), this implements a "weight tuning" behavior: mutating a small subset of weight genes perturbs the relative preference among transformation families, while mutating many weight genes can shift the overall transformation mix more drastically. It is also very important to mention that weights are allowed to become zero, enabling the GA to implicitly disable transformation families.

Overall, mutation allows of exploration in the continuous subspace (weights) while still being able to perform discrete jumps in the structural space (variant count and transformation depth). The intensity of the exploration is controlled through the mutation rate (percentage of mutated genes), which provides a direct parameter to trade off convergence speed with diversity preservation.

5.3.3. Crossover strategy

Offspring generation uses single-point crossover, for each mating event, two parents are selected (via tournament selection in the GA configuration) and recombined by choosing a single cut point along the chromosome, the prefix is inherited from one parent, and the suffix from the other. Given the chromosome layout, single-point crossover has two practically useful behaviors:

- If the cut happens early, offspring may inherit structural parameters from one parent and most of the weight profile from the other, resulting in combinations like "good

structural policy + good transformation mix".

- If the cut happens within the weight region, crossover blends weight profiles by exchanging contiguous blocks of weight genes, which can keep useful correlated patterns while still introducing novelty.

Validity is maintained because all genes remain in their declared domains at the GA level and decoding forces rounding and clamping for the discrete parameters, as a result, any offspring produced by crossover is immediately interpretable as a complete configuration.

5.3.4. Constraints handling

Constraints are incorporated as penalty terms [11] applied during fitness evaluation, this design keeps the search process flexible because the GA is allowed to explore configurations that temporarily violate constraints, but those violations are made progressively worse through explicit penalties that reduce the fitness score. The framework supports three classes of constraints:

- Global overhead budget: if an overhead budget is provided, configurations whose predicted total overhead exceeds the budget receive a penalty proportional to the violation.
- Minimum global protection target (optional): if a minimum acceptable protection level is specified, configurations whose worst-case protection across attacks falls below the target receive a penalty proportional to the violation.
- Per-attack protection targets (optional): the optimizer can also enforce attack-specific protection thresholds. When a per-attack target is provided, the configuration is penalized if the predicted protection for that specific attack is below the required level.

All constraint penalties are scaled by a configurable coefficient (`constraint_penalty`). As a result sufficiently large violations dominate the fitness, leading evolution toward feasible regions, while still allowing near-feasible candidates to compete when they offer better trade-offs [33].

5.4. Fitness function

The fitness function is designed to rank candidate configurations based on the trade-off between our two goals: security effectiveness against all three targeted TEA variants and performance to keep the execution overhead within the chosen range.

Because the GA maximizes a scalar score, the framework converts the multi-criteria evaluation into a single fitness value by predicting protection and overhead for each attack separately, then aggregating them into global metrics and finally applying a mode-dependent scalarization and optional constraint penalties [32].

Per attack evaluation

Evaluating a configuration by actually rewriting, compiling and measuring leakage and overhead for every GA individual would be very expensive. Because of this, the implementation uses a surrogate model trained from the empirical dataset (Runs.csv) [26][27]. For each attack type, a dedicated k-nearest-neighbors (KNN) regressor [12][4] is fitted to locally interpolate the measured points, each model takes as input the configuration features (structural parameters and transformation weights) and predicts two outputs:

- `cycles_overhead_pct` (performance cost)
- `bytes_stolen_pct` (leakage outcome)

To construct the input features, the configuration is encoded into a numerical vector: the discrete parameters are turned into continuous features (e.g. `num_variants` is log-scaled, `transforms_per_variant` is normalized) while the transformation weights are made attack-consistent by masking out transformations that are disabled for the current attack (via Mapping.csv), then normalizing the remaining enabled weights in order to form a correct distribution.

Aggregation across variants

Each candidate configuration must work across multiple threat models, not just one, for this reason, the global evaluation aggregates per-attack predictions into two global metrics:

- **Total overhead** (`overhead_total`): computed as the sum of predicted overhead percentages over the three attacks.
- **Worst-case protection** (`protection_min`): computed as the minimum protection among the three attacks, this is a conservative criterion since a configuration is only considered good if it does not leave any targeted TEA variant significantly weaker than the others.

In addition to overhead and protection, the evaluation also computes a "distance penalty"

that captures how far a candidate is located from the region supported by measured data. Concretely, the KNN model provides neighbor distances in the standardized feature space which is then converted to a linear penalty. This discourages the GA from exploiting unreliable extrapolations far away from the dataset, while still allowing controlled exploration.

Supported optimization modes

The implementation supports two practical optimization modes, corresponding to the two ways the problem is typically posed:

1. **Maximize protection under an overhead budget**, the fitness grows with worst-case protection and subtracts terms proportional to total overhead and distance penalty.
2. **Minimize overhead under a protection target**, the fitness is built as a maximization by negating total overhead (so lower overhead becomes higher fitness), while still slightly encouraging higher worst-case protection and subtracting the distance penalty.

This design ensures the GA optimizes a single configuration that is simultaneously effective against all three TEA variants, while explicitly controlling the overhead–protection trade-off through either an overhead threshold or a protection threshold, depending on the chosen optimization mode.

5.5. Why GA beats manual tuning

Manual tuning cannot be performed for this specific problem because the configuration space is both high-dimensional and non-linear, even restricting attention to the main parameters (number of variants, number of composed transformations, and a boolean switch), the outcome depends on the vector of transformation weights and changing it can alter both protection and overhead in ways that depend on the chosen variant count and transformation depth, making “one-parameter-at-a-time” tuning unreliable.

In addition, the objective optimized in this work is cross-attack: the goal is to optimize the worst-case protection across Spectre V1, Spectre V4, and Meltdown while respecting an overhead constraint (or minimizing overhead under a protection target). This "minimum across attacks" criterion makes the search unintuitive because improving one attack does not guarantee improving the overall score. A GA is well matched to this setting because it can explore combinations of structural parameters and weight vectors in parallel,

recombine promising partial solutions and search for robust trade-offs that are difficult to reach by hand.

5.6. Practical aspects

Computational cost is kept manageable by using surrogate-based evaluation since each candidate configuration is scored via the trained KNN models, rather than by performing full rewriting and measurement inside the GA loop.

Stopping criteria are implemented through standard GA limits (population size and number of generations), which directly bound the number of fitness evaluations, in practice, this makes the optimization procedure predictable in runtime.

6 | Experimental analysis

6.1. Experimental setup

The experiments were conducted in two different environments due to the hardware mitigations that are in place in modern processors which aggressively mitigate Meltdown. To evaluate the framework with Spectre V1 and Spectre V4 variants we used a 2019 MacBook Pro with Intel processor, while for Meltdown it was necessary to use a Linux OS running on a 2013 Intel processor which is vulnerable to it. Below is the detailed description of the environments:

- Mac machine:
  - OS version: macOS Sequoia 15.7.2
  - CPU model: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
  - Clang version: Apple clang version 17.0.0 (clang-1700.4.4.1) (Target: x86_64-apple-darwin24.6.0.)
- Linux machine:
  - OS version: Ubuntu 24.04.2 LTS
  - CPU model: Intel(R) Core(TM) i3-4130 CPU @ 3.40GHz
  - GCC version: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0

6.2. Attack PoCs

The experimental evaluation relies on three proof-of-concept (PoC) programs, one for each targeted TEA variant (Spectre V1, Spectre V4 and Meltdown). Each PoC is intentionally minimal and structured in a similar way: we have a transient gadget that makes a secret-dependent memory access and a Flush+Reload timing channel that reconstructs the secret by identifying which probe location was cached during transient execution.

6.2.1. Spectre V1 PoC

The Spectre V1 PoC is derived from the reference code distributed with the original Spectre paper [29] (as also stated in the header comments of the source [13]). The victim code performs a bounds check on an attacker-controlled index and, on the transient mispredicted path, uses the out-of-bounds byte to index a probe array (`array2[value * 512]`). The attacker logic repeatedly:

- flushes the probe array from cache (`clflush` over `array2[0..255]` with a fixed stride)
- trains the branch predictor with in-bounds indices and periodically injects the malicious index
- times accesses to the probe array using `rdtscp` and a cache-hit threshold, accumulating scores to select the most likely byte

The Spectre V1 PoC was taken from publicly available reference code [13]. [1]

6.2.2. Spectre V4 PoC

The Spectre V4 PoC implements a store-to-load forwarding/memory disambiguation scenario, the core idea is that the victim issues a store intended to overwrite the sensitive pointer target (writing `public_key`), followed immediately by a load that under speculative store bypass can transiently observe the stale value (the attacker-provided `secret_key`) and use it to index the probe array (`probe[secret_byte * 4096]`). In practice, the attacker loop:

- initializes the pointer chain so the victim will update `*memory_slot` to `public_key`, while the stale content points to `secret_key`
- flushes the pointer used by the victim and flushes all probe pages
- executes the victim function and then times the reload of `probe[i * 4096]` using `rdtscp`
- repeats many trials per byte and selects the character with the highest cache-hit count

This PoC is especially useful to highlight applicability constraints: depending on how the transient window is formed, some transformations may not have a meaningful placement

[1] https://github.com/crozone/SpectrePoC

within the vulnerable sequence without breaking semantics as we will see in Section 6.5.2. The Spectre V4 PoC was taken from publicly available reference code [34]. [2]

6.2.3. Meltdown PoC

The Meltdown PoC follows the classic "faulting load + dependent probe access" pattern, using a page-strided probe buffer mapped in user space. For each target byte at an input address (`start_addr + offset`), the PoC:

- flushes the 256 probe pages (`clflush`)
- transiently executes an illegal byte load (`movb (%ptr), %al`) and uses the loaded value to index the probe buffer (shift by 12 bits and load from `buf[value * page_size]`)
- recovers control after the fault via a `SIGSEGV` handler that redirects execution to a designated post-gadget label
- times reloads of all probe pages (serialized `rdtsc` sequence) and uses a small repetition/majority vote to pick the winning byte

The program supports dumping an arbitrary memory range (hex dump or raw output), which makes it a useful microbenchmark even when hardware performance counters are not uniformly available. The Meltdown PoC is adapted from publicly available reference code [16]. [3]

6.3. Evaluation metrics

This chapter evaluates each configuration with two metrics: **attack effectiveness** (how much information is recovered by the attacker) and **runtime cost** (how expensive the applied protection is). The same metrics are used both for reporting results and as inputs to the optimization stage.

Leakage effectiveness

For each PoC, the attacker attempts to reconstruct a fixed-length secret, we quantify leakage as the fraction of secret bytes that are correctly recovered:

$$\texttt{bytes\_stolen} = \sum_{i=0}^{B-1} \mathbf{1}\{\hat{s}_i = s_i\}$$

[2] https://github.com/mmxsrup/CVE-2018-3639
[3] https://github.com/Frichetten/meltdown-spectre-poc

$$\texttt{bytes\_stolen\_pct} = 100 \cdot \frac{\texttt{bytes\_stolen}}{B}$$

where s_i is the true secret byte, $\hat{s}_i$ is the byte inferred by the PoC and B is the secret length (40 bytes for Spectre V1, 16 bytes for Spectre V4, and 20 bytes for Meltdown in our setup). This metric captures exploitability in end-to-end fashion under the PoC's decoding strategy rather than only intermediate microarchitectural effects.

Performance cost (cycle overhead)

Runtime overhead is reported as percentage increase in CPU cycles with respect to the unprotected baseline of the same PoC on the same platform:

$$\texttt{cycles\_overhead\_pct} = 100 \cdot \frac{C_{\text{cfg}} - C_{\text{base}}}{C_{\text{base}}}$$

where C_{cfg} is the measured cycle count for a protected configuration and C_{base} is the cycle count of the corresponding unprotected build. Cycles are preferred over wall-clock time because they are less sensitive to frequency scaling and background system activity, while still reflecting the cost of additional fences, dependency chains and code expansion introduced by the transformations.

Auxiliary diagnostics (Spectre only)

For Spectre V1 and Spectre V4 we additionally collect branch mispredictions as secondary diagnostics to support interpretation of the overhead trends. These counters are used only as explanatory signals and are not treated as optimization objectives. For Meltdown, only leakage and cycle-based overhead are available in the recorded dataset. The branch misprediction overhead is computed as follows:

$$\texttt{branch\_mispred} = 100 \cdot \frac{M_{\text{cfg}} - M_{\text{base}}}{M_{\text{base}}}$$

Derived protection score (used by the optimizer)

In the optimization phase, leakage is also expressed as a protection score, so that higher is better and results remain comparable even when the unprotected PoC does not leak exactly 100% due to noise:

$$\texttt{protection\_pct} = 100 \cdot \left(1 - \frac{\texttt{bytes\_stolen\_pct}}{\texttt{bytes\_stolen\_pct}^{(\text{unprotected})}}\right)$$

clipped to [0,100], this yields 0% for the unprotected baseline and approaches 100% as leakage is suppressed.

6.4. Methodology and baselines

The experimental methodology follows an incremental approach where each step adds one protection mechanism on top of the previous one, so that both leakage reduction and overhead can be attributed to a specific design choice. Both the baselines and each one of the configurations included in the following sections was run fifty times and the results were obtained by computing the mean of the data.

Baseline definition and normalization

For each attack PoC, the reference point is the unprotected build, both metrics are normalized with respect to this baseline:

- **Leakage baseline**: bytes stolen by the unprotected PoC.
- **Overhead baseline**: cycles measured for the unprotected PoC, used to compute cycle overhead (%).

This ensures results are comparable across configurations within the same attack, independently of absolute runtime differences between PoCs and platforms.

6.4.1. Window duplication

The first protection step evaluates diversification in isolation, by enabling the multi-variant mechanism but duplicating the transient window without applying any transformation (all variants are functionally identical copies of the original window).

This baseline serves two purposes: it measures the "structural" cost of multi-variant execution (code duplication and runtime selection) and tests if diversification alone can already reduce leakage, due to perturbations in the attacker's control over the transient behavior. Configurations in this step vary primarily by the number of variants N, while keeping the window body unchanged.

6.4.2. Single-transformation (in isolation)

Starting from the same diversification settings used in window duplication (fixed N values), we then evaluate each compatible transformation in isolation, enabling exactly one

transformation at a time. The goal is to quantify the individual security/overhead contribution of each transformation before considering combinations.

Some transformations require a budget greater than one application per variant to become effective, for example, NOP-based perturbations are evaluated with `transforms_per_variant` > 1 to increase their impact. Compatibility is attack-dependent because only transformations that are applicable to the transient window structure and semantics-preserving for the specific PoC are considered in this step.

6.4.3. Hand-picked mixed configurations

In conclusion, we evaluate a small set of manually designed mixed policies, where multiple transformations are combined to maximize leakage reduction while maintaining overhead within a reasonable range. These configurations are "hand-picked", as the transformation mix and budgets are selected based on the behavior observed in single-transformation runs, rather than being produced by an optimizer.

6.5. Results per attack

In this section we will display the results for each of the targeted attack variants following the incremental approach described in Section 6.4.

6.5.1. Spectre V1

Table 6.1: Spectre V1: unprotected baseline and duplication-only (`same_variants=1`) configurations.

N	Bytes stolen (%)	Cycles overhead (%)	Branch mispred. (%)
1	97.8	0	0
5	93.9	15.48	1461.47
10	90.5	21.46	1616.04
25	89.4	26.8	1916.65
50	88.5	31.41	1864.44
75	86.8	32.69	1894.05
200	0	50.3	1922.98

Table 6.1 isolates the effect of diversification with no semantic changes to the transient window, even in this setting, leakage decreases monotonically as N grows (from $\approx$98% at $N = 1$ to $\approx$87% at $N = 75$), showing that variant multiplicity by itself already

perturbs the PoC's reliability. Interestingly, when increasing N even more the byte leakage starts to reduce drastically (probably due to saturation of the branch predictor) and it is removed completely for very high values of N such as 200. The corresponding overhead increases with N but remains in the ≈15–33% range (excluding when $N = 200$), providing a clear baseline for attributing additional gains to actual transformations. The branch misprediction % shows a large increase once window duplication is enabled, this is expected because Spectre V1 relies on branch predictor training and misprediction, so any disturbance of branch history and code layout can amplify miss events.

Table 6.2: Spectre V1: single-transformation baselines (one transformation family enabled at a time).

N	Transform	k	Bytes stolen (%)	Cycles overhead (%)
5	Fence	1	0	27.7
5	Idx	1	10.8	23.55
5	Lea	6	86.5	7.8
5	Nop	6	90.1	10.6
5	Reorder	6	87.6	17.9
10	Fence	1	0	28.6
10	Idx	1	6.89	23.68
10	Lea	6	86.1	16.5
10	Nop	6	89.3	17.66
10	Reorder	6	87.7	19.5
25	Fence	1	0	31.27
25	Idx	1	0	25.62
25	Lea	6	85.4	25.07
25	Nop	6	88.9	21.04
25	Reorder	6	87.5	24.37
50	Fence	1	0	33.47
50	Idx	1	0	27.73
50	Lea	6	83	27.01
50	Nop	6	85.1	14.81
50	Reorder	6	85.9	23.16
75	Fence	1	0	36.67
75	Idx	1	0	30.26
75	Lea	6	78.2	30.56
75	Nop	6	84.5	22.18
75	Reorder	6	79.5	27.63

In Table 6.2 each transformation family is enabled in isolation, allowing direct comparison of security–performance impact at fixed N. As expected, Fence eliminates leakage across all N values, but at a relatively high and stable overhead. Idx is particularly effective since

given moderate diversification it makes leakage go to (near) zero with lower overhead than full fencing, while Lea, Nop and Reorder provide only mild improvements over duplication-only, mostly trading performance for limited leakage reduction.

Table 6.3: Spectre V1: hand-picked mixed policies compared to duplication-only and fence-only baselines.

N	Policy	k	Mix	Bytes stolen (%)	Cycles overhead (%)
5	Duplication only	-	-	93.9	15.48
5	Fence-only	1	Fence 1	0	27.7
5	Hand-picked	6	Idx 0.3, Lea 0.3, Nop 0.2, Reord. 0.2	55.23	14.59
5	Hand-picked	6	Idx 0.3, Lea 0.25, Nop 0.2, Fence 0.05, Reord. 0.2	5.2	19.45
10	Duplication only	-	-	90.5	21.46
10	Fence-only	1	Fence 1	0	28.6
10	Hand-picked	6	Idx 0.3, Lea 0.3, Nop 0.2, Reord. 0.2	52.5	14.22
10	Hand-picked	6	Idx 0.3, Lea 0.25, Nop 0.2, Fence 0.05, Reord. 0.2	2.32	22.63
25	Duplication only	-	-	89.4	26.8
25	Fence-only	1	Fence 1	0	31.27
25	Hand-picked	6	Idx 0.3, Lea 0.3, Nop 0.2, Reord. 0.2	43.86	15.11
25	Hand-picked	6	Idx 0.3, Lea 0.25, Nop 0.2, Fence 0.05, Reord. 0.2	1.41	27.25
50	Duplication only	-	-	88.5	31.41
50	Fence-only	1	Fence 1	0	33.47
50	Hand-picked	6	Idx 0.3, Lea 0.3, Nop 0.2, Reord. 0.2	27.43	20.23
50	Hand-picked	6	Idx 0.3, Lea 0.25, Nop 0.2, Fence 0.05, Reord. 0.2	0	31.79
75	Duplication only	-	-	86.8	32.69
75	Fence-only	1	Fence 1	0	36.67
75	Hand-picked	6	Idx 0.3, Lea 0.3, Nop 0.2, Reord. 0.2	5.2	23.15
75	Hand-picked	6	Idx 0.3, Lea 0.25, Nop 0.2, Fence 0.05, Reord. 0.2	0	33.76

The hand-picked mixes outlined in Table 6.3 demonstrate that combining transforms can achieve better trade-offs than any single mechanism. In particular, the mixes with a small fence component lead leakage close to zero at overheads that are almost always lower than fence only, showing the benefit of distributing the mitigation across multiple perturbations rather than using exclusively on a barrier. The mixes which do not include fencing provide good improvements as N grows, but they confirm that for Spectre V1 some barrier-like component is typically needed to reliably remove leakage.

6.5.2. Spectre V4

Spectre V4 exposes a very compact store–load transient window, as a consequence, some transformations cannot be injected without violating applicability constraints, in those cases, the corresponding configuration effectively becomes equivalent to the duplication-only baseline.

Table 6.4: Spectre V4: unprotected baseline and duplication-only (`same_variants=1`) configurations.

N	Bytes stolen (%)	Cycles overhead (%)	Branch mispred. (%)
1	100	0	0
100	96.53	4.87	-30.8
250	88.28	8.34	-21.5
500	75.35	10.23	-40.56
1000	55.63	16.02	-42.32
2500	15.89	25.28	-47.92
5000	0	48.34	-55.98

In Spectre V4, diversification-only shows a much stronger dependence on the number of variants (as shown in Table 6.4): leakage decreases gradually as N increases, reaching 0% only at very large N (5000), but with a correspondingly large overhead (≈48%). This indicates that for speculative store bypass, the PoC remains robust under duplication, and meaningful leakage reduction via diversification alone requires extreme multi-variant expansion. This table therefore sets expectations that V4 will typically need either stronger transforms or higher N to reach low leakage values.

Interestingly, branch mispredictions decrease (negative deltas) as N grows, this is consistent with the attack signal becoming weaker because as fewer probe lines are transiently cached, the probe-loop condition (`time2 <= threshold`) becomes more and more biased towards the "not-taken" path, which the branch predictor can learn more reliably. Therefore, branch-miss results here should be interpreted as an indirect consequence of reduced cache-hit activity during probing.

Table 6.5: Spectre V4: single-transformation baselines (one transformation family enabled at a time). Transforms marked with † are not applicable inside the transient window and therefore coincide with duplication-only.

N	Transform	k	Bytes stolen (%)	Cycles overhead (%)
100	Fence	1	33.42	10.78
100	Idx†	6	96.5	4.87
100	Lea	6	95.32	5.23
100	Nop	6	96.48	4.97
100	Reorder†	6	96.5	4.87
100	SSB	1	45.23	8.69
250	Fence	1	22.78	14.14
250	Idx†	6	88.28	8.34
250	Lea	6	87.21	9.87
250	Nop	6	88.59	8.91
250	Reorder†	6	88.28	8.34
250	SSB	1	31.79	11.93
500	Fence	1	7.82	17.65
500	Idx†	6	75.35	10.23
500	Lea	6	74.32	11.12
500	Nop	6	75.21	10.11
500	Reorder†	6	75.35	10.23
500	SSB	1	12.21	15.76
1000	Fence	1	0	24.53
1000	Idx†	6	55.63	16.02
1000	Lea	6	53.18	23.53
1000	Nop	6	55.7	16.92
1000	Reorder†	6	55.63	16.02
1000	SSB	1	4.58	20.89
2500	Fence	1	0	37.02
2500	Idx†	6	15.89	25.28
2500	Lea	6	8.32	26.79
2500	Nop	6	15.89	25.32
2500	Reorder†	6	15.89	25.28
2500	SSB	1	0	34.93

Table 6.5 shows that, unlike Spectre V1, several transformations have limited or no effect in V4 because they are not applicable inside the relevant transient window (marked †, they are equivalent to duplication-only). Among applicable defenses, Fence and SSB provide important leakage reductions already at moderate N at the cost of higher overhead than duplication-only. Lea and Nop produce small improvements, suggesting that small structural perturbations are insufficient against the V4 gadget unless they directly constrain speculation or enforce ordering.

Differently from Spectre V1, inserting a fence does not immediately result in zero leakage for small N. That is because in this PoC the exploitable behavior is a speculative store bypass performed in a tight store–load sequence and mitigation effectiveness is inherently probabilistic, in fact even when speculation is strongly constrained, residual transient executions (and finite-sample noise in Flush+Reload decoding) can still lead to a non-zero number of correctly recovered bytes.

The overhead of fencing is also lower with respect to Spectre V1 because the overall runtime of each attempt is dominated by the measurement harness (flushing and timing 256 probe pages for every try), so the fence cost is amortized over a large amount of work outside the transient gadget. Furthermore, the reference PoC already executes an `_mm_mfence()` in the attacker loop before invoking the victim, meaning the baseline already includes frequent serialization, so the incremental penalty of adding one more barrier inside the victim window is smaller than in scenarios where the baseline is fully unfenced (like the Spectre V1 PoC). This explains why the leakage reaches 0% only once N is sufficiently large, where diversification further reduces the residual success probability.

The mixed policy shown in Table 6.6 improves over duplication-only, it does not outperform the strongest single-barrier baselines in leakage terms, but it provides a better security–overhead trade-off for sure. This reflects the fact that for V4 the effective mitigation tends to be dominated by mechanisms which enforce ordering (Fence/SSB), while simpler transformations contribute only to second-order effects. The table shows a middle ground: moderate overhead with substantial leakage reduction (but not complete removal), this becomes relevant when an overhead budget rules out fencing everywhere.

Table 6.6: Spectre V4: hand-picked mixed policy compared to duplication-only and fence-only baselines.

N	Policy	k	Mix	Bytes stolen (%)	Cycles overhead (%)
100	Duplication only	-	-	96.53	4.87
100	Fence-only	1	Fence 1	33.42	10.78
100	Hand-picked	6	SSB 0.2, Fence 0.2, Lea 0.3, Nop 0.3	78.35	9.26
250	Duplication only	-	-	88.28	8.34
250	Fence-only	1	Fence 1	22.78	14.14
250	Hand-picked	6	SSB 0.2, Fence 0.2, Lea 0.3, Nop 0.3	64.97	13.67
500	Duplication only	-	-	75.35	10.23
500	Fence-only	1	Fence 1	7.82	17.65
500	Hand-picked	6	SSB 0.2, Fence 0.2, Lea 0.3, Nop 0.3	35.29	15.69
1000	Duplication only	-	-	55.63	16.02
1000	Fence-only	1	Fence 1	0	24.53
1000	Hand-picked	6	SSB 0.2, Fence 0.2, Lea 0.3, Nop 0.3	25.25	19.03
2500	Duplication only	-	-	15.89	25.28
2500	Fence-only	1	Fence 1	0	37.02
2500	Hand-picked	6	SSB 0.2, Fence 0.2, Lea 0.3, Nop 0.3	12.43	29.21

6.5.3. Meltdown

For Meltdown, Table 6.7 displays that duplication-only provides only a modest reduction in bytes stolen as N increases, while overhead remains relatively small. This implies that diversification by itself does not significantly disrupt the transient gadget, the attack remains very effective unless stronger countermeasures are introduced.

Table 6.7: Meltdown: unprotected baseline and duplication-only (`same_variants=1`) configurations.

N	Bytes stolen (%)	Cycles overhead (%)
1	100	0
100	97.51	3.32
250	96.74	5.49
500	93.98	5.83
1000	90.89	6.43
2500	86.95	8.16
5000	83.21	11.28

Table 6.8: Meltdown: single-transformation baselines (one transformation family enabled at a time).

N	Transform	k	Bytes stolen (%)	Cycles overhead (%)
100	Fence	1	1.67	25.49
100	Lea	3	95.94	5.46
100	Nop	6	97.31	2.45
100	Sand	1	10.89	14.37
250	Fence	1	0	27.49
250	Lea	3	95.25	6.57
250	Nop	6	96.88	5.47
250	Sand	1	5.21	16.76
500	Fence	1	0	29.15
500	Lea	3	92.47	5.82
500	Nop	6	93.71	7.06
500	Sand	1	1.23	18.80
1000	Fence	1	0	32.76
1000	Lea	3	89.24	6.56
1000	Nop	6	90.35	6.36
1000	Sand	1	0	21.28
2500	Fence	1	0	35.90
2500	Lea	3	86.02	8.48
2500	Nop	6	86.48	7.78
2500	Sand	1	0	23.35

By looking at Table 6.8 we can clearly separate simple perturbations from mitigation mechanisms that directly break the disclosure path. FenceCut and Pointer Sandboxing (Sand) are the only options that consistently drive leakage to (near) zero at moderate N, even though with substantial overhead, while Lea and Nop remain close to duplication-only in both leakage and cost.

Table 6.9: Meltdown: hand-picked mixed policies compared to duplication-only and FenceCut-only baselines.

N	Policy	k	Mix	Bytes stolen (%)	Cycles overhead (%)
100	Duplication only	-	-	97.51	3.32
100	Fence-only	1	Fence 1	1.67	25.49
100	Hand-picked	4	Nop 0.25, Lea 0.30, Fence 0.15, Sand 0.30	12.88	16.74
100	Hand-picked	6	Nop 0.30, Lea 0.40, Fence 0.05, Sand 0.25	8.34	18.24
250	Duplication only	-	-	96.74	5.49
250	Fence-only	1	Fence 1	0	27.49
250	Hand-picked	4	Nop 0.25, Lea 0.30, Fence 0.15, Sand 0.30	10.23	17.79
250	Hand-picked	6	Nop 0.30, Lea 0.40, Fence 0.05, Sand 0.25	6.21	18.54
500	Duplication only	-	-	93.98	5.83
500	Fence-only	1	Fence 1	0	29.15
500	Hand-picked	4	Nop 0.25, Lea 0.30, Fence 0.15, Sand 0.30	3.92	20.05
500	Hand-picked	6	Nop 0.30, Lea 0.40, Fence 0.05, Sand 0.25	5.81	21.56
1000	Duplication only	-	-	90.89	6.43
1000	Fence-only	1	Fence 1	0	32.76
1000	Hand-picked	4	Nop 0.25, Lea 0.30, Fence 0.15, Sand 0.30	6.76	23.07
1000	Hand-picked	6	Nop 0.30, Lea 0.40, Fence 0.05, Sand 0.25	1.34	24.57
2500	Duplication only	-	-	86.95	8.16
2500	Fence-only	1	Fence 1	0	35.90
2500	Hand-picked	4	Nop 0.25, Lea 0.30, Fence 0.15, Sand 0.30	4.58	26.08
2500	Hand-picked	6	Nop 0.30, Lea 0.40, Fence 0.05, Sand 0.25	0	27.59

The hand-picked mixes shown in Table 6.9 explore if partial use of strong defenses (Cut/Sand) combined with simple perturbations can reduce leakage while containing overhead. At smaller N, the mixes generally trade some residual leakage for lower overhead

compared to Cut-only, while at larger N they can approach zero leakage with overhead below the pure fence baseline. Overall, the table outlines the same theme as the other attacks: mixing transforms can improve the security–performance trade-off, but only if the mix includes at least one mechanism that directly targets the primary cause of leakage.

6.6. Genetic optimization results

This section reports the results produced by the genetic optimizer described in Chapter 5, which searches the configuration space automatically and selects a single global configuration meant to be applied consistently across Spectre V1, Spectre V4 and Meltdown.

The optimizer reports projected metrics, configurations are evaluated through the surrogate-based fitness function (built from the experimental dataset) and the final output is a predicted average overhead and attack protection for each of the three. Table 6.10 summarizes two representative optimization queries: a budget-driven run and a security-driven run.

Table 6.10: Genetic optimizer: selected configurations under two constraint profiles.

Scenario	N	k	Transform mix	Avg. ovh. (%)	V1 prot. (%)	V4 prot. (%)	M prot. (%)
Max overhead (24%)	1027	1	Sand=0.2288, Idx=0.1894, Lea=0.1821, SSB=0.1402, Fence(SL)=0.1353, Reord=0.0585, Nop=0.0368, Fence(Cut)=0.0197, Fence(JCC)=0.0093	24.02	95.74	95.93	96.86
Min protection (96.5%)	3287	1	Sand=0.2597, Fence(SL)=0.1630, SSB=0.1388, Reord=0.1333, Nop=0.1063, Fence(JCC)=0.0901, Idx=0.0765, Fence(Cut)=0.0232, Lea=0.0092	25.90	99.26	96.49	97.48

Table 6.10 highlights two consistent patterns. First, the optimizer tends to select a large number of variants (N in the ∼1000–3000 range) while keeping k small, implying that at these trade-off points it is more convenient to diversify instead of applying long transfor-

mation chains for each variant. Second, the final mixes are multi-mechanism because they always combine at least one ordering or speculation-stopping transformation (SSB barrier and/or store–load fence) with simple perturbating and data-hardening ones (index masking, pointer sandboxing). This is consistent with the per-attack results, where Spectre V4 is usually the limiting case and benefits most from ordering-enforcing mechanisms.

6.6.1. Comparison vs fixed per-attack baselines

To make the genetic optimizer results comparable with the manual campaign, we select, for each attack individually, two strong fixed baselines from Section 6.5:

- **best hand-picked configuration**, intended as the lowest-overhead setting that achieves a protection level similar to the optimizer output
- **best fence-everywhere configuration**, intended as the lowest-overhead setting that achieves maximal (or near-maximal) protection by relying only on fence-like transformations

Table 6.11: GA (24% budget) vs best per-attack baselines. For GA, overhead is the optimizer's projected average across attacks (24.02%) and per-attack protection is projected by the surrogate fitness pipeline.

Attack	Strategy	N	k	Protection (%)	Overhead (%)
Spectre V1	GA (global, 24% budget)	1027	1	95.74	24.02
	Best hand-picked (per-attack)	10	6	97.68	22.63
	Best fence-everywhere (per-attack)	5	1	100.00	27.70
Spectre V4	GA (global, 24% budget)	1027	1	95.93	24.02
	Best hand-picked (per-attack)	2500	6	87.57	29.21
	Best fence-everywhere (per-attack)	1000	1	100.00	24.53
Meltdown	GA (global, 24% budget)	1027	1	96.86	24.02
	Best hand-picked (per-attack)	1000	6	98.66	24.57
	Best fence-everywhere (per-attack)	100	1	98.33	25.49

Table 6.11 compares these baselines with the GA solution obtained under a 24% maximum overhead budget. Note that for hand-picked and fence-everywhere rows, overheads are measured in Section 6.5, while the reported protection is derived from bytes-stolen (also taken from Section 6.5). The comparison highlights three points.

First, fence-oriented strategies provide the highest protection in absolute terms, but they always suffer from higher overhead than the GA solution (most clearly for Spectre V1 and marginally for Spectre V4 and Meltdown). Under a strict overhead budget, this

supports the claim that the optimizer is better than "fencing everything": it distributes the available overhead across multiple transformations (masking, dependency barriers, pointer sandboxing and fences) instead of relying exclusively on fence-like primitives.

Second, the GA configuration is competitive with the best hand-picked settings on Spectre V1 and Meltdown, sacrificing only a few percentage points of protection to stay within the budget. More importantly, on Spectre V4 the GA solution is much better than the best hand-picked baseline, obtaining substantially higher protection (95.93% vs 87.57%) at lower overhead (24.02% vs 29.21%). This is consistent with the per-attack analysis where V4 is the limiting case and benefits most from mixes that include ordering or speculation-stopping components.

Third, hand-picked and fence-everywhere results implicitly assume attack-specific selection: one would deploy different mitigation choices depending on whether the threat is V1, V4 or Meltdown. In contrast, the genetic optimizer produces a **single global configuration** that achieves high protection across all attacks simultaneously, in one pass of the framework, without requiring the user to know in advance which TEA variant is relevant and which mitigation family is the best match. This property, which is the crucial part of the framework, becomes increasingly valuable if the approach is extended to additional TEA variants, where manual tuning rapidly becomes impractical.

Pareto frontier analysis

The optimization problem is intrinsically bi-objective, if we increase diversification and apply stronger transformations protection will improve but so will the overhead. In this context, a configuration is *Pareto-optimal* if no other configuration achieves both equal-or-higher minimum protection and equal-or-lower overhead, with at least one strict improvement, the set of all Pareto-optimal points forms the (approximate) *Pareto frontier* [14]. Even when the GA is executed in a single-objective form (by turning constraints into penalties), the resulting best solutions can be interpreted as samples from this frontier. Table 6.12 reports three representative solutions that illustrate the trade-off.

Table 6.12: Representative GA solutions showing the protection–overhead trade-off (projected metrics).

Run profile	N	Avg. ovh. (%)	Min prot. (%)
Max overhead (23%)	1060	22.80	91.12
Max overhead (24%)	1027	24.02	95.74
Min protection (96.5%)	3287	25.90	96.49

A key observation is that the worst-case protection is typically given by Spectre V4: when the overhead budget is reduced, the first attack to lose protection is V4, while V1 and Meltdown remain similarly easier to mitigate. On the contrary, enforcing a higher minimum protection shifts the selected Pareto points towards a larger number of variants (larger N) and a heavier importance on ordering or speculation-stopping transformations.

6.7. Security and limitations analysis

This section consolidates the main limitations of the proposed prototype and the key threats to validity that may affect the interpretation and generalization of the experimental results reported in this chapter. The goal is to clarify what security claims are supported by the evaluation under the assumed threat model, and which factors may limit coverage, effectiveness, or portability.

6.7.1. Security scope and interpretation of results

The framework is designed as a software-only, compile-time hardening stage for x86-64 compiler-emitted assembly. The security objective is to reduce exploitable transient behavior for the targeted TEA classes (Spectre V1, Spectre V4, and Meltdown-like patterns) by selectively rewriting candidate transient windows and diversifying their implementation. As a consequence, the results presented in this chapter should be interpreted as evidence that the framework can expose and navigate a meaningful security–overhead trade-off space under explicit cross-attack constraints, rather than as a universal guarantee of mitigation effectiveness for arbitrary TEA variants or all microarchitectures.

6.7.2. Measurement noise and metric scope

Transient-execution PoCs are inherently noisy: observed leakage depends on predictor state, store/load timing, cache contention, interrupts, OS scheduling, and other sources of microarchitectural variability. Even with repeated runs and aggregation over a fixed

secret, residual variability can affect both bytes stolen and cycle counts, especially when leakage is already low and close to the decoding threshold. In addition, the chosen metrics have limitations: bytes stolen captures end-to-end PoC success, but partially reflects the stability of the specific decoding strategy; cycles provide a convenient proxy for runtime cost, but do not capture other relevant overhead dimensions such as code size or instruction-cache pressure.

6.7.3. External validity and platform dependence

The evaluation relies on three microbenchmark PoCs and a limited set of x86-64 environments. PoCs isolate the transient gadget and enable controlled comparisons, but they do not represent complex application workloads; both mitigation effectiveness and overhead may therefore change with larger codebases, different speculation contexts, and different input distributions. Moreover, TEA behavior and mitigation impact are microarchitecture-dependent: speculation, forwarding behavior, and cache hierarchy details influence both leakage and the effectiveness of specific transformations. For these reasons, results should not be treated as uniform across all processors.

6.7.4. Coverage and applicability limitations

Not all transformations are applicable to all transient windows, Spectre V4 is representative: the exploitable region is a compact store–load window and many transformations cannot be applied under the framework safety constraints, effectively reducing to duplication-only. More generally, detection and rewriting are based on conservative pattern matching performed over compiler-generated assembly, therefore uncommon instruction sequences, aggressive optimizations, or complex control flow may reduce the coverage or restrict semantics-preserving modifications. Finally, the evaluation targets Spectre V1, Spectre V4, and Meltdown and does not claim coverage of all TEA variants.

6.7.5. Diversification and optimization caveats

Many strong configurations (both manual and GA-selected) use high variant multiplicity (often on the order of 10^3), which increases binary size, build time, and instruction-cache pressure, these costs are not fully captured by cycles alone. In addition, GA results are obtained through a dataset-driven fitness pipeline, which can introduce surrogate error and dataset bias. GA-selected configurations should therefore be treated as strong candidates and ideally re-validated through direct execution on the target platforms and workloads before deployment decisions are made.

6.7.6. Prototype limitations

While the prototype aims to be robust for assembly written by the compiler, several limitations follow from its scope and its intentionally simple analysis.

Heuristic analysis at the assembly level: window identification is pattern-based and does not construct full control-flow or dataflow models, as a consequence, the detectors may miss gadgets (false negatives) or mark benign code (false positives) and it cannot guess high-level properties like if a load will fault.
ISA and dialect dependence: the current implementation is specialized for x86-64 compiler output in AT&T syntax and assumes typical GCC/Clang formatting conventions, porting to other ISAs or assembler dialects would require reworking both parsing and transformation patterns.
Limited semantic reasoning about memory: the tool cannot guarantee that transformations that rewrite address computations are safe for all memory-access corner cases, the implementation mitigates this by using conservative applicability checks and skipping transformations if the operand structure does not match supported patterns, but this still limits coverage.
Microarchitecture-dependent effectiveness: several transformations aim to alter speculative behavior and microarchitectural leakage, which depend on the target CPU's speculation, forwarding, and caching details, the tool can guarantee that the rewritten code is structurally different, but it cannot guarantee a uniform reduction in leakage across all processors and configurations.
Code-size and performance overhead: diversification is achieved by duplicating windows and including additional code, which increases code size and may introduce nontrivial overhead, especially when many windows are detected or when configurations with high number of variants and/or many transformations per variant are selected.

These limitations are consistent with the tool's role as a research prototype. It wants to provide a practical mechanism for experimenting with diversification and local protection strategies at the assembly level while leaving stronger guarantees and further portability as directions for future work.

7 | Conclusions and future developments

This thesis investigated how to make software mitigations for Transient Execution Attacks (TEAs) more scalable and tunable, by moving from a single hardening policy to a configurable space of assembly-level transformations and by using automated search to select good security–performance operating points.

7.1. Summary of contributions

The main outcomes of this work can be summarized as follows:

- **Assembly-to-assembly mitigation framework**: a compile-time tool that rewrites x86-64 compiler-emitted assembly by detecting candidate transient windows for Spectre V1, Spectre V4 (SSB) and Meltdown-like patterns and selectively hardening only the affected regions.
- **Variant-based hardening with lightweight runtime selection**: for each detected window, the framework injects a small selector and multiple semantically equivalent variants (including the original), enabling both diversification (reduced gadget repeatability) and security–overhead trade-offs through configurable mixes of local transformations.
- **Transformation catalog including multiple mitigation families**: the implementation provides composable transformations which cover speculation-stopping, data sanitization/hardening, dependency injection and structural perturbation, with conservative applicability checks used to preserve correctness.
- **Genetic optimization across multiple attacks**: since the configuration surface grows quickly, a genetic optimizer is used to select a single global configuration that targets Spectre V1, Spectre V4 and Meltdown simultaneously under either an overhead budget or a minimum-protection constraint exposing an (approximate)

Pareto trade-off between protection and overhead.

- **Experimental evaluation and key empirical observations**: the evaluation on three PoCs shows that mixing mitigation mechanisms can improve the trade-off with respect to single-family baselines, Spectre V4 tends to be the limiting case as budgets tighten, and GA-selected configurations can outperform hand-picked settings under cross-attack constraints.

7.2. Future work

The prototype demonstrates that diversified selective rewriting paired with automated search is a practical way to navigate TEA mitigation trade-offs, but several improvements are possible given the current limitations:

- **Integration into LLVM pipeline**: re-implement the pipeline as an LLVM-based pass to operate before final assembly emission, leveraging LLVM analyses to reduce false positives/negatives.
- **Broader TEA coverage**: extend the detector set beyond V1, V4 and Meltdown-like patterns, both by adding new pattern families and by revisiting the threat model to include additional transient triggers and gadget shapes. This should go together with a broader set of PoCs and validation scenarios.
- **Portability to other ISA and dialects.** Generalize parsing and rewriting beyond x86-64 AT&T syntax (e.g. Intel syntax first, then other ISAs), a possibility could be to replace the ad-hoc parser with a backend built on existing assembler/disassembler components and then re-define detectors and transformations in an ISA-parametric way.
- **Additional validation on realistic workloads and microarchitectures**: evaluate the approach on larger application-level codebases and across more microarchitectures, since both mitigation effectiveness and overhead are known to vary with speculation contexts and CPU details. This also enables studying secondary costs like binary size and build time under high variant multiplicity, which become important for deployability.
- **Attack-aware transformation design for constrained windows**: Spectre V4 highlights that many transformations are inapplicable under current safety constraints, effectively reducing to duplication-only in several cases, designing additional, window-compatible transformations (or relaxing constraints safely with a

stronger analysis) would improve worst-case protection and reduce the need for very large N.

Bibliography

[1] Intel Analysis of Speculative Execution Side Channels. Technical report, 2018. URL www.intel.com/design/literature.htm.

[2] *Ultimate SLH: Taking Speculative Load Hardening to the Next Level*. USENIX Association, 2023. ISBN 9781939133373.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: an efficient, scalable alternative to reorder buffers. 2003.

[4] N. S. Altman. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3):175–185, 8 1992. ISSN 0003-1305. doi: 10.1080/00031305.1992.10475879.

[5] K. Arikan, A. Palumbo, L. Cassano, P. Reviriego, S. Pontarelli, G. Bianchi, O. Ergin, and M. Ottavi. Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(7):938–951, 7 2022. ISSN 15579999. doi: 10.1109/TVLSI.2022.3171810.

[6] J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, and N. Zeldovich. Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract. Technical report.

[7] T. Blickle and L. Thiele. A Comparison of Selection Schemes Used in Evolutionary Algorithms. *Evolutionary Computation*, 4(4):361–394, 12 1996. ISSN 1063-6560. doi: 10.1162/evco.1996.4.4.361.

[8] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry. WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly. 1 2024. doi: 10.1016/j.cose.2024.103731. URL http://arxiv.org/abs/2309.07638http://dx.doi.org/10.1016/j.cose.2024.103731.

[9] C. Canella, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, J. Van Bulck, F. Piessens, D. Evtyushkin, and D. Gruss. *A Systematic Evaluation of Transient Execution At-*

tacks and Defenses. ISBN 9781939133069. URL `www.usenix.org/conference/usenixsecurity19/presentation/canella`.

[10] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction using Store Sets. Technical report.

[11] C. A. Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, 1 2002. ISSN 00457825. doi: 10.1016/S0045-7825(01)00323-1.

[12] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1 1967. ISSN 0018-9448. doi: 10.1109/TIT.1967.1053964.

[13] crozone. Spectre V1 Proof of Concept. URL `https://github.com/crozone/SpectrePoC`.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6 (2):182–197, 4 2002. ISSN 1089778X. doi: 10.1109/4235.996017.

[15] L. Fiolhais and L. Sousa. Transient-Execution Attacks: A Computer Architect Perspective. *ACM Computing Surveys*, 56(3), 3 2024. ISSN 15577341. doi: 10.1145/3603619.

[16] Frichetten. Meltdown Proof of Concept. URL `https://github.com/Frichetten/meltdown-spectre-poc`.

[17] A. F. Gad. PyGAD: An Intuitive Genetic Algorithm Python Library. 6 2021. URL `http://arxiv.org/abs/2106.06158`.

[18] J. Ge and F. Zhang. FlushTime: Towards Mitigating Flush-based Cache Attacks via Collaborating Flush Instructions and Timers on ARMv8-A. Technical report.

[19] L. Gerhorst, H. Herzog, P. Wägemann, M. Ott, R. Kapitza, and T. Hönig. VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions. In *ACM International Conference Proceeding Series*, pages 644–659. Association for Computing Machinery, 9 2024. ISBN 9798400709593. doi: 10.1145/3678890.3678907.

[20] A. Godbole, Y. A. Manerkar, and S. A. Seshia. SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security. In *CCS 2024 - Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communica-*

tions Security, pages 2756–2770. Association for Computing Machinery, Inc, 12 2024. ISBN 9798400706363. doi: 10.1145/3658644.3690214.

[21] D. E. Goldberg and J. H. Holland. Genetic Algorithms and Machine Learning. *Machine Learning*, 3(2-3):95–99, 10 1988. ISSN 0885-6125. doi: 10.1023/A:1022602019183.

[22] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. *Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks*. ISBN 978-1-939133-04-5. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/gras`.

[23] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. Technical report. URL `https://github.com/IAIK/KAISER`.

[24] J. H. Holland. Genetic Algorithms and the Optimal Allocation of Trials. *SIAM Journal on Computing*, 2(2):88–105, 6 1973. ISSN 0097-5397. doi: 10.1137/0202009.

[25] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. Technical report.

[26] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 6 2011. ISSN 22106502. doi: 10.1016/j.swevo.2011.05.001.

[27] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, 13(4):455–492, 12 1998. ISSN 0925-5001. doi: 10.1023/A:1008306431147.

[28] R. E. Kessler. Many Forms of Out-of-Order and Speculative Execution, and a High-Bandwidth Memory System. Technical report, 1999.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. 1 2018. URL `http://arxiv.org/abs/1801.01203`.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. *Meltdown: Reading Kernel Memory from User Space*. ISBN 978-1-939133-04-5. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/lipp`.

[31] G. Maisuradze and C. Rossow. ret2spec. In *Proceedings of the 2018 ACM SIGSAC*

Conference on Computer and Communications Security, pages 2109–2122, New York, NY, USA, 10 2018. ACM. ISBN 9781450356930. doi: 10.1145/3243734.3243761.

[32] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 4 2004. ISSN 1615-147X. doi: 10.1007/s00158-003-0368-6.

[33] Z. Michalewicz and M. Schoenauer. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation*, 4(1):1–32, 3 1996. ISSN 1063-6560. doi: 10.1162/evco.1996.4.1.1.

[34] mmxsup. Spectre V4 Proof of Concept. URL `https://github.com/mmxsrup/CVE-2018-3639`.

[35] N. Mosier, J. C. Mitchell, and C. Trippel. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. Technical report. URL `https://github.com/nmosier/llsct`.

[36] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, D. Stefan, and S. Diego. Swivel: Hardening WebAssembly against Spectre. Technical report. URL `https://swivel.pro`.

[37] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. 10 2018. URL `http://arxiv.org/abs/1805.08506`.

[38] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. Technical report, 2005.

[39] A. Seznec and P. Michaud. A case for (partially) TAgged Geometric history length branch prediction.

[40] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. *ACM SIGARCH Computer Architecture News*, 13(3):36–44, 6 1985. ISSN 0163-5964. doi: 10.1145/327070.327125.

[41] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann, and M. Jan. µARCHIFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections. *Formal Methods in Computer-Aided Design*, 2023. doi: 10.34727/2023/isbn.978-3-85448-060-0. URL `https://doi.org/10.34727/2023/isbn.978-3-85448-060-0`.

[42] G. Wang. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. Technical report. URL `https://github.com/winter2020/oo7`.

[43] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2), 6 1994. ISSN 0960-3174. doi: 10.1007/BF00175354.

[44] Y. Yarom and K. Falkner. *Proceedings of the seventeenth Large Installation Systems Administration Conference (LISA XVII) : October 26-31, 2003 San Diego, CA, USA*. USENIX Association, 2003. ISBN 9781931971157.

A | Appendix A

A.1. Parsing example

Listing A.1: Parsing example

```
.text
.globl victim
.type victim, @function
victim:
    .cfi_startproc
    pushq %rbp
    movq %rsp, %rbp # prologue

LBB0_2:
    movq (%rdi,%rcx,8), %rax
    cmpq $0, %rax
    jne Lreturn
    .cfi_def_cfa_offset 16

Lreturn:
    popq %rbp
    ret
    .cfi_endproc

.size victim, .-victim
```

How it is stored internally (this is a conceptual view, not a literal printout):

- Function container victim
  - Header directives (kept as raw strings, in order):
    - .text
    - .globl victim

    * .type victim, @function

- Body as an ordered list of line records (each record preserves the raw line; instructions also expose mnemonic/operands/comment):

    1. Directive line record

        * raw: .cfi_startproc

        * classified as "directive inside function" (kept in-place)

    2. Instruction record, raw: pushq %rbp, mnemonic: pushq, operands: [%rbp]

    3. Instruction record with comment, raw: movq %rsp, %rbp # prologue, mnemonic: movq, operands: [%rsp, %rbp], comment: prologue

    4. Internal label record (treated as a raw line inside the current function, not a new function), raw: LBB0_2:

    5. Instruction record showcasing operand splitting with parentheses, raw: movq (%rdi,%rcx,8), %rax, mnemonic: movq, operands: [(%rdi,%rcx,8), %rax], (note: the parenthesized addressing mode stays a single operand token)

    6. Instruction record, raw: cmpq $0, %rax, mnemonic: cmpq, operands: [$0, %rax]

    7. Instruction record (control-flow), raw: jne Lreturn, mnemonic: jne, operands: [Lreturn]

    8. Directive line record inside function, raw: .cfi_def_cfa_offset 16, preserved verbatim to avoid breaking unwind/debug info

    9. Internal label record, raw: Lreturn:

    10. Instruction record, raw: popq %rbp, mnemonic: popq, operands: [%rbp]

    11. Instruction record, raw: ret, mnemonic: ret, operands: [ ]

    12. Directive line record that also closes the function scope, raw: .cfi_endproc, preserved in-place; after this line the parser considers the function terminated

- Trailing-directives container (synthetic scope used only to preserve remaining directives)

    * header directives:

- .size victim, .-victim

* no function label is emitted for this synthetic container; it exists only to keep directive order intact in the output

The parser keeps a faithful ordered stream of lines per function, but upgrades real instructions into structured records (mnemonic + operands + optional comment) so that later stages (detectors and rewriting) can operate on them without losing the ability to reproduce the original assembly layout.

A.2. Variant generation example

Listing A.2: Variant generating example, original code.

```
## Start of the transient window
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
## End of the transient window
    shll $9, %eax
    ...
```

Listing A.3: Variant generating example, after code injection.

```
## RANDOM SELECTOR BLOCK
    subq $32, %rsp
    pushq %rax
    pushq %rcx
    pushq %rdx
    rdtsc
    xorq %rdx, %rax
    movl $3, %ecx
    xorl %edx, %edx
    divl %ecx
    movl %edx, %eax
    popq %rdx
    popq %rcx
    cmpl $0, %eax
    je .Lvictim_function_win0_var0
```

```
    cmpl $1, %eax
    je .Lvictim_function_win0_var1
    cmpl $2, %eax
    je .Lvictim_function_win0_var2
    jmp .Lvictim_function_win0_var0
## Variant 0
.Lvictim_function_win0_var0:
    popq %rax
    addq $32, %rsp
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
    jmp .Lvictim_function_win0_continue
## Variant 1
.Lvictim_function_win0_var1:
    popq %rax
    addq $32, %rsp
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
    jmp .Lvictim_function_win0_continue
## Variant 2
.Lvictim_function_win0_var2:
    popq %rax
    addq $32, %rsp
    cmpq %rcx, %rax
    jae LBB0_2
    movq -8(%rbp), %rcx
    leaq _array1(%rip), %rax
    movzbl (%rax,%rcx), %eax
    jmp .Lvictim_function_win0_continue
.Lvictim_function_win0_continue:
    shll $9, %eax
    ...
```

List of Figures

Listings

List of Tables

Acknowledgements